# Improving Byzantine Protocols with Secure Computational Components

Miguel Correia, Alysson N. Bessani,
Nuno F. Neves, Lau C. Lung, Paulo Veríssimo

DI–FCUL TR–05–20

December 2005

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

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

# Improving Byzantine Protocols with Secure Computational Components*

**Miguel Correia**† **Alysson N. Bessani**‡ **Nuno F. Neves**† **Lau C. Lung**§ **Paulo Veríssimo**†

† Faculdade de Ciências da Universidade de Lisboa, Bloco C6, Piso 3, Campo Grande, 1749-016 Lisboa – Portugal

‡ Departamento de Automação e Sistemas, Universidade Federal de Santa Catarina – Brasil

§ Programa de Pós-Graduação em Informática Aplicada, Pontifícia Universidade Católica do Paraná – Brasil

**Abstract**

Byzantine-tolerant protocols are currently being used as building blocks in the construction of secure applications, therefore their performance has a practical impact. Work in message-passing distributed protocols typically considers a set of nodes interconnected by a network. This paper investigates the benefits for the performance of Byzantine-tolerant protocols of including a secure component in the nodes. We have been exploring this kind of hybrid fault models by calling these subsystems *wormholes*. The present paper follows this line but considers local wormholes, while in previous work wormholes were distributed, i.e., they included their own communication channel. The paper presents the architecture of systems with local wormholes and several flavors of consensus based on this model. The paper also presents the first work with asynchronous wormholes, using randomization to circumvent FLP, while also providing the first formalization of wormholes-based protocols using I/O automata. The benefits of the approach are discussed.

## 1 Introduction

The development of efficient distributed protocols has both theoretical and practical interest. Today, Byzantine-tolerant protocols are being used as important building blocks in the construction of secure applications based on a recent approach: *intrusion tolerance* [30]. This approach can be considered to be part of the ongoing effort to make computing systems more secure, Internet included, vis-a-vis the large number of security incidents permanently reported by entities like CERT/CC[1].

Work in message-passing distributed protocols typically considers a set of nodes, running a software component called process, interconnected by a network. Here, we consider distributed systems prone to Byzantine faults, including malicious faults. Work in the area assumes that processes can fail in a Byzantine way (violate the protocol in any possible way) and that the network can corrupt the communication, e.g., by dropping, modifying or repeating messages [2, 3, 4, 9, 10, 11, 20, 25]. This system architecture is depicted in Figure 1(a). In terms of timeliness, these systems are usually considered to be mostly asynchronous, but extended with some oracle [2, 10, 9, 20] or time assumption [11] to circumvent the Fischer, Lynch and Paterson (FLP) impossibility result [13]. This result can also be circumvent using randomization [3, 4, 25] (a survey of early work is in [6]).

This paper investigates the benefits for the performance of Byzantine-tolerant protocols of making this picture slightly more complicated. Suppose each node now includes a second component $w$ that can communicate both with the process $p$ (locally) and with similar components in other nodes (through the network). This architecture is shown in Figure 1(b). Notice that we are not saying at this stage what is $w$: it might be either hardware or software. However, $w$ has an important characteristic: it can only fail by crashing (fail-stop), not in a Byzantine way, i.e., it is secure or tamperproof. Therefore, we have a *hybrid fault model*: nodes include parts that can only fail by crashing ($w$) and parts that can fail arbitrarily, or in a Byzantine way (everything except $w$)[2]. Each of these oracles includes a random oracle, i.e., a random number generator. We do not consider any other oracles or time assumptions.

*This work was partially supported by the FCT through project POSI/EIA/60334/2004 (RITAS) and the Large-Scale Informatic Systems Laboratory (LASIGE).

[1] http://www.cert.org

[2] This kind of fault model is clearly different from some previous work in hybrid fault models, starting in [21], in which fault distributions are simply assumed. Here we design the secure component with the purpose of enforcing its fault model, an obvious requirement in environments prone to malicious faults.

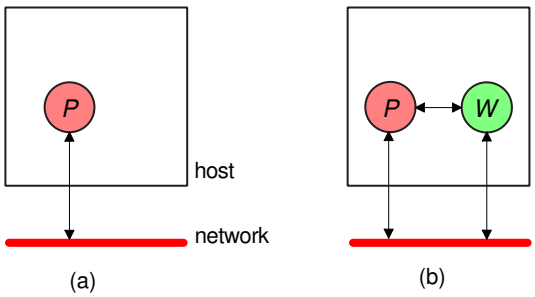


Figure 1: (a) Typical system architecture. (b) System architecture we explore in the paper.

Two concerns may reasonably be raised about this model. First: is it possible to implement such a model or is it of purely theoretical interest? The answer is simple: there are many ways of implementing $w$. Most computers, either laptops, desktops or servers can be extended with some kind of secure hardware that can be used to run $w$, e.g., USB tokens, Smartcards, secure coprocessors and PC/104 appliances[3]. Software solutions include running $w$ as a special process in a security kernel.

The second problem is: is this model relevant? Why cannot we just make $p$ secure, i.e., $p = w$, and end up with a fail-stop model? The first answer is that sometimes it is possible to do so and a system designer should consider that possibility. However, in general $p$ is part of a complex application with millions of lines of code that cannot be put inside $w$ (USB tokens and Smartcards have very limited resources) and cannot even be secured in that way because it has complex interactions with its environment, e.g., with people and networked services. If we want to make $w$ secure or fail-stop, it has to satisfy two properties derived from the classical reference monitor properties [15][4]:

- Isolation. The wormhole must be tamperproof or secure. This is considered to be an assumption throughout the paper, although it has to be enforced in an implementation.
- Verifiability. Its security has to be formally verifiable. This is true for the instances of $w$ we present in the paper, since they implement reasonably simple distributed protocols.

The issue explored in the paper is: what are the benefits for the performance of Byzantine-tolerant protocols of the model in Figure 1(b)? Is there any interest for such protocols of having a secure component in the nodes?

**Context and related work.** We have been exploring this kind of hybrid fault models by calling these subsystems *wormholes* [28]. The metaphor comes from an astrophysics concept that some Science Fiction has presented as shortcuts that might be used to travel fast to faraway places in the Universe[5]. The idea we have been exploring is to take advantage of components with stronger properties to handle some kind of uncertainty. The first work in this line used a distributed real-time wormhole to handle uncertainty in terms of time [29]. Afterwards, a distributed real-time and secure wormhole was used to build Byzantine fault-tolerant protocols, i.e., to handle uncertainty in terms of malicious faults [8, 7, 23].

The present paper follows this work on wormholes but has an important difference: here the wormholes are components *existing locally inside the nodes*, while in previous work wormholes were distributed. These distributed wormholes included not only local components in the nodes, but these components were interconnected by a dedicated communication channel or network. Therefore, in this paper we simplify the architecture by removing this channel/newtork. We use randomized oracles to avoid synchrony assumptions about the network.

Several security protocols have been previously proposed that use different types of (local) secure components to prevent intrusions in critical modules. Tygar and Yee show how a secure coprocessor can be used, e.g., to guarantee the security of an electronic payment scheme [27]. Itoi and Honeyman use smartcards for secure

[3]Several cryptographic modules that might be used with this purpose were validated for conformance to FIPS PUB 140-1 and FIPS PUB 140-2 (Security Requirements for Cryptographic Modules) by the National Institute of Standards and Technology. A list is available at: http://csrc.nist.gov/cryptval/140-1/1401val.htm.

[4]The third property that a reference monitor must satisfy is specific for access control (completeness).

[5]See, e.g., http://en.wikipedia.org/wiki/Wormhole

authentication with Kerberos [17]. Shoup and Rubin use also smartcards to enhance the security of session key distribution [26]. Avoine and colleagues present an algorithm for deterministic fair exchange also based on secure components [1]. Several other examples might be listed. However, all these works use secure components with the purpose of ensuring some of the security properties of the protocols. To the best of our knowledge this paper is the first that uses local secure components with the purpose of *improving the performance of distributed systems algorithms* like consensus. Here the purpose is not to prevent intrusions in certain components and ensure certain security properties, but to improve the performance of protocols that tolerate intrusions in some of the nodes.

**Paper results.** The contributions of the paper are the following:

- it presents the architecture of systems with local wormholes (the possibility of using local wormholes was envisaged when the concept was introduced but never explored [24, 28]);
- it discusses the benefits of distributed algorithms based on local wormholes and presents consensus protocols of several flavors with time-complexities similar to the complexities of fail-stop consensus protocols: binary consensus, multi-valued consensus and vector consensus; it also provides the first formalization of wormhole-based protocols using I/O automata;
- it presents the first work with strictly asynchronous wormholes, using randomization to circumvent FLP.

# 2 System Model

We formalize the system using I/O automata, a formalism introduced by Lynch and Tuttle [19, 18]. In this formalism, system components are modelled by I/O automata. An automaton receives input actions and generates output and internal actions. A system is represented by a composition of automata.

The system we consider in the paper is asynchronous, i.e., we assume no bounds on processing and communication delays. There is a set of $n$ processes $\Pi = \{p_1, p_2, ...p_n\}$ and a set of $n$ wormholes $\Upsilon = \{w_1, w_2, ...w_n\}$. Each node $i$ contains a process $p_i$ that can access the wormhole $w_i$ (see Figure 1(b)).

Each wormhole includes a random oracle module. This oracle provides random numbers from a finite set $\mathcal{U}$ with uniform distribution. We postpone the discussion about the content of $\mathcal{U}$ to Section 2.2.

## 2.1 Fault Model

The architecture we are considering is more complex than what is commonly considered in message-passing distributed algorithms so there is also more to be said about the fault model.

A process is said to be *correct* if it does not *fail* during the execution of the protocol, i.e., if it follows the protocol. If a process fails it is said to be *corrupt* or *failed*. We use the letter $f$ to denote the maximum number of processes assumed to fail during an execution of the protocol.

A process can fail in the usual Byzantine ways, for instance: it can stop, delay the communication, send spurious messages, or transmit several messages with the same identifier. Corrupt processes can pursue a plan of breaking the properties of the protocol alone or in collusion with other failed processes.

There are also new modes of failure that are architecture related. A wormhole is also said to be *correct* if it does not fail, i.e., if it does not *crash*. Otherwise it is said to be *crashed* or *failed*. In addition to the situations listed before, a process $p_i$ can fail if $w_i$ crashes, if $p_i$ does not manage to communicate with $w_i$ for some reason (e.g., because an attacker controls the node) or if its communication with $w_i$ is modified in some way.

Typically all processes would be connected by communication channels, but in the simple protocols we present in the paper processes do not communicate directly but only through the wormholes, so we do not need to make any statement about these channels. The wormholes are fully-connected by reliable channels with two properties: if the sender and the recipient of a message are both correct then (1) the message is eventually received and (2) the message is not modified in the channel. In practice, these channels can be implemented in common LANs or the Internet using secure communication protocols such as the Secure Socket Layer [14]. Notice that the assumption of reliable channels is a way of hiding (masking) the failures in the channels: message modifications, replays, omissions and spurious messages. The communication can be delayed arbitrarily, but all messages are eventually delivered correctly.

## 2.2 Formal System Model

Each process $p_i$ is modelled as an automaton with five actions (see Figure 2):

- input $propose(v)_i$ – invocation of the protocol by user $U_i$;
- output $decide(v)_i$ – response to user $U_i$ with the value decided by the protocol;
- output $w\_call(v)_i$ – passage of a value to the wormhole $w_i$;
- input $w\_resp(v)_i$ – response from the wormhole $w_i$;
- input $byz\_failure_i$ – signals the Byzantine failure of the automaton.

The user $U_i$ represents the application that calls process $p_i$. The automata composition in the figure represents the system that executes the protocols we are going to describe, therefore it does not include that user. However, $U_i$ might also be modelled as an automaton.

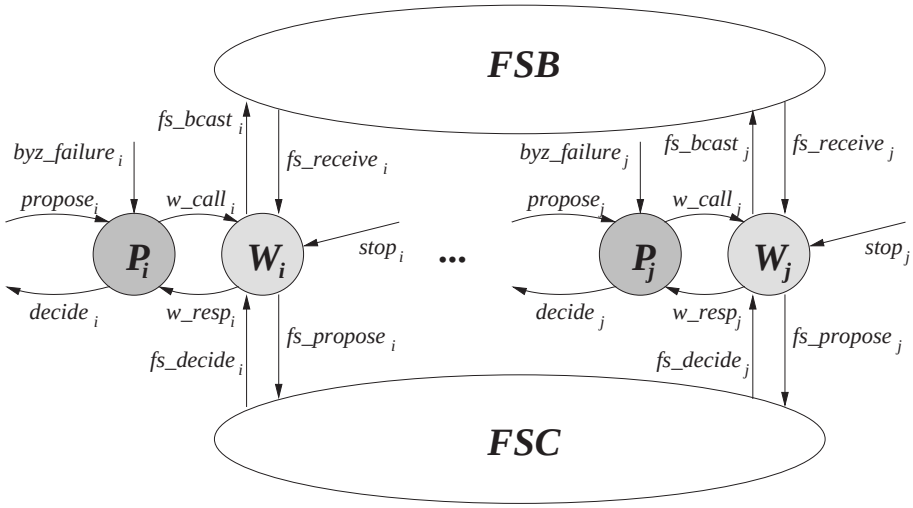


Figure 2: Automata composition for all protocols in the paper. The automaton $p_i$ models process $p_i$, automaton $w_i$ models wormhole $w_i$, automaton $FSB$ a fail-stop broadcast, and $FSC$ a fail-stop consensus.

Wormhole $w_i$ is modelled as an automaton with several actions, some of which are (see the same figure):

- input $w\_call(v)_i$ – corresponds to the output with the same name in process $p_i$;
- output $w\_resp(v)_i$ – corresponds to the input with the same name in process $p_i$;
- input $stop_i$ – signals the crash of the wormhole.

The composition includes a broadcast channel FSB with a semantics equivalent to the sender wormhole individually sending the message to all wormholes in $\Upsilon$ (including itself) using the reliable channels presented in the previous section. The primitive is used by the wormholes that are fail-stop, therefore all recipients receive the same message unless a wormhole crashes. The signature is the following:

- input $fs\_bcast(v)_i$ – send the value $v$ to every wormhole $w_i \in \Upsilon$;
- output $fs\_receive(j, v)_i$ – receive a value $v$ from process $j$.

The signature includes two actions related to the fail-stop consensus automaton $FSC$, which we describe in the next section: $fs\_propose(v)_i$ and $fs\_decide(v)_i$.

The failures of wormholes (crash) and processes (Byzantine) are modelled by inputs with distinct meanings. The input $stop_i$ is the usual way of modelling the crash of an automaton [18] and is handled explicitly in the code of the automaton (see Algorithm 2). The use of an input $byz\_failure_i$ to model Byzantine failures was first suggested in [5]. When this event occurs, the automaton is substituted by another automaton with the same signature but with arbitrary behavior.

**Consensus Among Wormholes**

The protocols we present in the paper use as building block a consensus protocol executed by the wormholes. This protocol does not need to tolerate Byzantine faults since the wormholes are assumed to be fail-stop and fully-connected by reliable channels (Section 2.1), therefore the protocol is essentially a *fail-stop consensus*. The protocol circumvents FLP using the above mentioned random oracle modules.

We model the fail-stop consensus as a single automaton FSC (Figure 2). In terms of system architecture, this automaton models part of the behavior of the wormholes (the fail-stop consensus) plus the reliable channels connecting the wormholes. Therefore, in reality, any wormhole $w_i$ is modelled by the automata $w_i$ *and* FSC. The objective of modelling the fail-stop consensus as an automaton separate from the wormholes is to have modularity, thus allowing us to plug-in different consensus modules into our algorithms.

The problem of consensus is, informally, the problem of making a set of entities (processes, wormholes) agree on a common value. A wormhole $w_i$ is said to *propose* a value $v \in \mathcal{V}$ for an execution of the consensus protocol when an output action $fs\_propose(v)_i$ occurs in $w_i$. The wormhole is said to *decide* on a value $v$ when an input action $fs\_decide(v)_i$ occurs in $w_i$. Consensus is formally defined in terms of the following properties:

- *Validity-1:* If a correct wormholes decides $v$, then $v$ was proposed by some wormhole.
- *Agreement:* No two correct wormholes decide differently.
- *Termination:* Every correct wormhole eventually decides with probability 1.

In the paper we use two variants of the consensus protocol: binary consensus ($\mathcal{V} \equiv \{0, 1\}$) and multi-valued consensus ($\mathcal{V}$ is a finite set of values). An example of a fail-stop binary consensus protocol is presented in [3], while a fail-stop multi-valued consensus can be found in [12]. The random oracle used in the former provides values in the set $\mathcal{U} \equiv \{0, 1\}$, while in the latter provides values in $\mathcal{U} \equiv \{1, 2, ..., n\}$ (where $n$ is the total number of processes). A transformation from binary to multi-valued fail-stop consensus is presented in [22]. All these protocols tolerate the failure of at most half less one processes/wormholes ($f = \lfloor \frac{n-1}{2} \rfloor$).

# 3 Byzantine Consensus

Consensus is an important distributed systems problem since it can be used as the main building block to solve several other agreement problems [16, 9]. Several protocols for Byzantine consensus in asynchronous systems have been proposed, using several methods to circumvent FLP: randomization [3, 25], failure detectors [20, 2], partial-synchrony [11] and distributed wormholes [7].

The *resilience* of a distributed protocol is the maximum number of failed processes it can tolerate. The maximum resilience for Byzantine consensus in asynchronous systems is $f = \lfloor \frac{n-1}{3} \rfloor$ out of a total of $n$ processes [4, 11], which is also the resilience of the protocols we propose in the paper.

Let us consider the same definition of consensus given in Section 2.2 (with 'processes' instead of 'wormholes') but with a different Validity property (this is the typical definition used in the literature [11, 20, 7]):

- *Validity-2.* If all correct processes propose the same value $v$, then any correct process that decides, decides $v$.

Our Byzantine consensus protocol solves both *binary and multi-valued consensus* if we instance FSC respectively with a binary or a multi-valued fail-stop consensus. A direct consequence of the system architecture depicted in Figure 1(b) is that the protocol is executed both in the processes and the wormholes (respectively $p$ and $w$ in the figure). The code executed by the processes and wormholes is presented respectively in Algorithms 1 and 2.

The presentation of the protocol assumes a property of well-formedness, both for the users that call the protocol ($U_i$) and for the processes ($p_i$) [18]:

- *Well-formedness.* For any $i$, the interactions between $U_i$ and $p_i$, and the interactions between $p_i$ and $w_i$, are *well-formed* for $i$.

**Algorithm 1** Consensus protocol (process $p_i$)

**Signature:**
Input: $propose(v)_i, w\_resp(V)_i, byz\_failure_i$
Output: $w\_call(v)_i, decide(v)_i, decide(\perp)_i$
$v \in \mathcal{V}$, $\perp \notin \mathcal{V}$

**State:**
$prop = \perp$, value proposed by the user
$Vect = \perp$, vector with several proposed values
$failed = false$, true if the process is corrupt

**Transitions:**

1: input $propose(v)_i$
2: Eff: $prop \leftarrow v$

3: input $w\_resp(V)_i$
4: Eff: $Vect \leftarrow V$

5: input $byz\_failure_i$
6: Eff: $failed \leftarrow true$

7: output $w\_call(v)_i$
8: Pre: $prop = v$
9: Eff: $prop \leftarrow \perp$

10: output $decide(v)_i$
11: Pre: $\#_v(Vect) \geq f + 1$
12: Eff: $Vect \leftarrow \perp$

13: output $decide(\perp)_i$
14: Pre: $\#_v(Vect) < f + 1$
15: Eff: $Vect \leftarrow \perp$

**Algorithm 2** Consensus protocol (wormhole $w_i$)

**Signature:**
Input: $w\_call(v)_i, fs\_receive(j, v)_i, fs\_decide(v)_i, stop_i$
Output: $fs\_propose(v)_i, fs\_bcast(v)_i, w\_resp(v)_i$
$v \in \mathcal{V}$, $\perp \notin \mathcal{V}$

**State:**
$prop = \perp$, value proposed by the process
$dec = \perp$, vector decided by FSC
$\forall j \in \Pi : Vect[j] = \perp$, vector with values delivered by FSB
$stopped = false$, true if the wormhole stopped

**Transitions:**

1: input $w\_call(v)_i$
2: Eff: $prop \leftarrow v$

3: input $fs\_receive(j, v)_i$
4: Eff: $Vect[j] \leftarrow v$

5: input $fs\_decide(Vect)_i$
6: Eff: $dec \leftarrow Vect$

7: input $stop_i$
8: Eff: $stopped \leftarrow true$

9: output $fs\_bcast(v)_i$
10: Pre: $\neg stopped \wedge prop = v$
11: Eff: $prop \leftarrow \perp$

12: output $fs\_propose(Vect)_i$
13: Pre: $\neg stopped \wedge \#_\perp(Vect) \leq f$
14: Eff: $\forall j \in \Pi : Vect \leftarrow \perp$

15: output $w\_resp(Vect)_i$
16: Pre: $\neg stopped \wedge dec = Vect$
17: Eff: $dec \leftarrow \perp$

Let us consider the interaction between the user $U_i$ and the automaton $p_i$. A sequence of actions $propose(v)_i$ and $decide(v)_i$ is said to be *well-formed* for $i$ if it is some prefix of the cyclically ordered sequence $propose(v')_i$, $decide(v'')_i, propose(v''')_i, decide(v'''')_i, ...$ This property essentially excludes the possibility of a user making two proposals before the decision of the protocol is returned. The objective of making this assumption is to make Algorithm 1 more simple, by not having to consider explicitly the case of ill-formed interactions. Nevertheless, this assumption might be discarded with simple modifications to the algorithm, like identifying each consensus execution with a consensus id (*cid*), and substituting the variables *prop* and *dec* by sets containing tuples $(cid, v)$ for the active consensuses. Similar considerations might be done about the well-formedness of the interactions between $p_i$ and $w_i$.

The protocol is very simple[6] and follows the typical format for I/O automata protocols [19]. They start with the declaration of the automata signature, i.e., its input and output actions. Then, they declare the state variables and the transitions corresponding to each action, specified in terms of preconditions (*Pre:*) and effects (*Eff:*).

In the protocol, vectors have one entry per process in $\Pi$ (or wormhole in $\Upsilon$) and are designated by an uppercase letter. The function $\#_x(Vect)$ counts the number of occurrences of $x$ in vector $Vect$. In Algorithm 1, line 11, this function is used to select a value $v$ that occurs at least $f + 1$ times in $Vect$. If there are two values $v_1$ and $v_2$ in that condition, the function returns the one that appears first in $Vect$.

The proof that the protocol solves the consensus problem is independent of the consensus being binary or

[6]Except for the fail-stop consensus executed by the wormholes, which is not displayed (several protocols can be used).

multi-valued:

**Theorem 1** *If at most $f = \lfloor \frac{n-1}{3} \rfloor$ processes are failed, then the protocol specified by Algorithms 1 and 2 solves consensus as specified by properties Validity-2, Agreement and Termination.*

*Proof (sketch):* The protocol is based on a consensus protocol executed by the wormholes. This protocol is defined in terms of the properties in Section 2.2. The properties are satisfied if no more than $\lfloor \frac{n-1}{2} \rfloor$ out of $n$ wormholes fail, an immediate consequence of the assumption than no more than $\lfloor \frac{n-1}{3} \rfloor$ processes fail (Section 2.1).

*Validity-2.* When a value is proposed, process $p_i$ gives it to the wormhole $w_i$ (Alg.1:1-2,7-9)[7] that sends it to every other wormhole, including itself (Alg.2:1-2,9-11). Then, the wormhole waits for $(n - f)$ messages with values proposed by different wormholes (Alg.2:3-4,12-14). Wormholes are fail-stop so they either send the message once, or do not send it at all. At most $f$ processes can fail, therefore at least $f + 1$ messages come from correct processes: $(f = \lfloor \frac{n-1}{3} \rfloor) \Rightarrow (n - 2f \geq f + 1)$. The property Validity-2 assumes all correct processes propose $v$, therefore all vectors given to FSC contain at least $f + 1$ copies of $v$ (Alg.2:12-14). The vector decided by FSC, which is one of the vectors proposed, is returned to the process (Alg.2:5-6,15-17), that returns the value that appears at least $f + 1$ times in the vector, i.e., $v$ (Alg.1:3-4,13-15). When the correct processes do not propose the same value, the protocol decides a default value $\perp$ (Alg.1:13-15).

*Agreement.* The proof derives trivially from the fact that all non-crashed wormholes return the same vector (Alg.2:5-6,15-17), which is used to decide deterministically the value returned (Alg.1:3-4,10-15). Note that the function $\#$ is deterministic even if there are two values $v_1$ and $v_2$ such that $\#_{v_1}(Vect) = \#_{v_2}(Vect) \geq f + 1$. In that case, the value among $v_1$ and $v_2$ that appears first in $Vect$ is returned.

*Termination.* An inspection of the two algorithms shows that the protocol terminates if two conditions are satisfied. The first is that at least $n - f$ wormholes have to broadcast the values proposed by the corresponding processes (Alg.2:13). This must happen since at least that number of processes are correct. The second condition is that the FSC consensus has to terminate, something that is guaranteed by the property of Termination of that protocol (Section 2.2). □

### 3.1 Vector Consensus

*Vector consensus* is a variant of the problem of consensus, which is specially interesting when Byzantine faults are considered [10, 2, 23]. A vector consensus protocol instead of deciding a value, returns a vector. This vector has values proposed by a majority of correct processes, something that can be useful to solve practical distributed system problems, like atomic multicast [9]. The definition is the same as for the consensus protocols above, except for the Validity property:

- *Validity-VC.* 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
  2. at least $f + 1$ elements of the vector $Vect$ are the initial values of correct processes.

A protocol that solves vector consensus is presented in Algorithms 3 (process automaton) and 2 (wormhole automaton, same as for binary/multi-valued consensus). The protocol is a trivial modification of the previous multi-valued consensus. It simply returns the vector decided by the wormholes, instead of choosing the most frequent value in the vector. The automaton FSC has also to be instantiated with a multi-valued fail-stop consensus.

**Algorithm 3** Vector consensus protocol (process $p_i$)

*Everything identical to Algorithm 1 except lines 10-15 that are substituted by:*

10: output $decide(Vect)_i$
11: Pre: $Vect \neq \perp$
12: Eff: $Vect \leftarrow \perp$

[7] We use this short notation to reference Algorithm 1, lines 1 to 2 and 7 to 9.

**Theorem 2** *If at most $f = \lfloor \frac{n-1}{3} \rfloor$ processes are failed, then the protocol specified by Algorithms 3 and 2 solves vector consensus as specified by properties Validity-VC, Agreement and Termination.*

*Proof (sketch):* The protocol is very similar to the consensus protocol, so the proofs of Agreement and Termination are also the same as in Theorem 1.

*Validity-VC. Property 1.* If $p_i$ is correct then $w_i$ gets the initial value $v$ (Alg.3:1-2,7-8; Alg.2:1-2), which it broadcasts to all wormholes in $\Upsilon$ (Alg.2:9-11). Every wormhole proposes to FSC a vector with the default value ($\perp$) or the broadcasted value $v$ (Alg.2:3-4) in the entry corresponding to $p_i$. FSC simply decides on one of the vectors proposed by the wormholes, and this is the vector decided by the protocol (Alg.2:5-6,15-16; Alg.3:3-4,10-12) therefore the property is satisfied.

*Validity-VC. Property 2.* We proved that the vector decided is one of the vectors proposed to FSC by one of the wormholes. These vectors include at least $n - f$ entries filled (Alg.2:12-14) and at most $f$ of these entries contain values from failed processes. The property is satisfied since $f = \lfloor \frac{n-1}{3} \rfloor \Rightarrow n - f - f \geq f + 1$. □

# 4 Evaluation of the Protocols

Randomized Byzantine agreement protocols are usually evaluated in terms of resilience, and time and communication complexities [6]. *Time complexity* in asynchronous systems is normally measured by counting the number of asynchronous rounds. In this kind of protocols, an asynchronous round is defined in the following way: a process usually broadcasts a message to all other processes, and then waits for $(n - f)$ messages broadcasted by the others in the same round; when it gets that number of replies, it either goes to the next round or terminates. For randomized protocols, the metric is usually the *expected* number of asynchronous rounds, since the number of rounds can only be defined probabilistically. We evaluate the protocols in the situation where the failed processes do the best they can to delay the protocol, the network behaves the worst as possible and the initial values are also the worst possible combination. *Communication complexity* can be measured in number of bits sent (per round or protocol execution) or in number of messages sent. Here we use the expected number of message broadcasts.

| | **Consensus type** | **Fault model** | **Oracle** | **Resilience** | **Expected Time complexity** | **Expected Communication complexity** | **Reference** |
|---|---|---|---|---|---|---|---|
| 1 | binary | crash | random | $\lfloor \frac{n-1}{2} \rfloor$ | $2^{n-1} + 1$ | $(2^{n-1} + 1)n$ | [3] |
| 2 | binary | crash | random | $\lfloor \frac{n-1}{3} \rfloor$ | $1.5 \times 2^{n-f-1} + 2.5$ | $(1.5 \times 2^{n-f-1} + 2.5)n$ | §A |
| 3 | multi-val. | crash | random | $\lfloor \frac{n-1}{2} \rfloor$ | $2^{n-1} + 2$ | $(2^{n-1} + 1)n + n^2$ | [3]+[22] |
| 4 | multi-val. | crash | random | $\lfloor \frac{n-1}{3} \rfloor$ | $1.5 \times 2^{n-f-1} + 3.5$ | $(1.5 \times 2^{n-f-1} + 2.5)n + n^2$ | §A+[22] |
| 5 | multi-val. | crash | random | $\lfloor \frac{n-1}{2} \rfloor$ | $n^{n-1} + 2$ | $(n^{n-1} + 1)n + 2n^2$ | [12] |
| 6 | binary | Byz. | random | $\lfloor \frac{n-1}{5} \rfloor$ | $2^{n-f-1} + 1$ | $(2^{n-f-1} + 1)n$ | [3] |
| 7 | binary | Byz. | random | $\lfloor \frac{n-1}{3} \rfloor$ | $4.5(2^{n-f-1} + 1)$ | $(2^{n-f-1} + 1)3n^2$ | [4] |
| *8* | *binary* | *Byz.* | *wormh.* | $\lfloor \frac{n-1}{3} \rfloor$ | $1.5 \times 2^{n-f-1} + 3.5$ | $(1.5 \times 2^{n-f-1} + 3.5)n$ | §3+§A |
| *9* | *multi-val.* | *Byz.* | *wormh.* | $\lfloor \frac{n-1}{3} \rfloor$ | $1.5 \times 2^{n-f-1} + 4.5$ | $(1.5 \times 2^{n-f-1} + 4.5)n + n^2$ | §3+§A+[22] |
| *10* | *vector* | *Byz.* | *wormh.* | $\lfloor \frac{n-1}{3} \rfloor$ | $1.5 \times 2^{n-f-1} + 4.5$ | $(1.5 \times 2^{n-f-1} + 4.5)n + n^2$ | §3.1+§A+[22] |

Table 1: Comparison of several asynchronous randomized consensus protocols.

Table 1 compares our protocols with several other asynchronous randomized consensus protocols that have been published previously. Randomized consensus protocols are essentially of two kinds: (1) those based on local random oracle modules, following Ben-Or's seminal paper [3]; (2) those based on a secret-sharing scheme that distributes identical random numbers to all processes, starting with Rabin's work [25]. The second class includes protocols that run in less rounds at the cost of expensive public-key cryptography and having to distribute shares of random numbers before the execution of the protocol. We do not consider a secret-sharing scheme in the paper so our protocols fit in the first class and we compare them only with protocols in that class.

Each row in the table is about one protocol. The top rows (1-5) evaluate fail-stop consensus protocols, the middle rows (6-7) evaluate Byzantine consensuses in the literature, and the bottom rows (8-10) our own protocols. Our protocols need a binary or a multi-valued fail-stop consensus (Section 2.2). For reasons we explain below, instead of the binary protocol in [3] (row 1) we use a modified – fail-stop – version of the Byzantine protocol

in [4] (see Appendix A). The multi-valued protocol is this same binary consensus combined with the 'binary to multi-valued' transformation in [22] (see the column Reference in the table; § indicates the section or appendix where our protocol is described).

The first conclusion we take from the table is that the time and message complexities of our protocols are similar to the best complexities of fail-stop protocols. Compare the binary consensuses in rows 2 and 8 and the multi-valued consensuses in rows 4 and 9.

The second conclusion is that these complexities are better than those of Byzantine resilient protocols in the literature. Comparing our binary protocol (row 8) to Bracha's protocol [4] (row 7) we see that the time complexities are both $O(2^{n-f})$ (although ours is slightly lower) but our communication complexity is clearly lower: $O(2^{n-f}n+n)$ against $O(2^{n-f}n^2+n^2)$. The time and communication complexities of [3] are apparently identical to ours, respectively $O(2^{n-f})$ and $O(2^{n-f}n+n)$. However, the resilience of that protocol is suboptimal (only $\lfloor\frac{n-1}{5}\rfloor$ out of $n$) so if we wanted to tolerate the same number of faults the complexities would be considerably worse (exponential with a base greater than 2). We did not find multi-valued or vector consensuses of the class we are considering in the literature, so we cannot make a comparison for those protocols.

The reason why we used a modified version of Bracha's protocol instead of Ben-Or's protocol [3] to evaluate our protocols can now be understood. Ben-Or's protocol tolerates $f=\lfloor\frac{n-1}{2}\rfloor$ crashes (row 1), the optimal resilience for fail-stop protocols. The time and communication complexities are respectively $O(2^n)$ and $O(2^n n+n)$. The resilience of the protocol is more than we need for our binary consensus protocol, but its complexities would lead to similar complexities for our protocols. The problem is that a time complexity of $O(2^n)$ is worse than the complexities of current Byzantine-resilient protocols (rows 6-7). The same is true for the communication complexity. Using the modified version of Bracha's protocol (Appendix A) we manage to have better complexities: $O(2^{n-f})$ and $O(2^{n-f}n+n)$. The resilience of this modified protocol is suboptimal for fail-stop protocols but exactly what we need since it is used to support a Byzantine protocol.

One final comment is that the exponential time complexities we obtained are not particularly good for consensus protocols. However, our point here is to compare protocols with local wormholes with protocols based on the usual architecture. Better complexities might probably be obtained using Rabin's aproach, but this was left as future work.

# 5 Conclusion

The need for more secure distributed systems is raising a renewed interest in efficient Byzantine protocols. This paper investigates the contribution for that objective of including a secure component – local wormhole – inside the system nodes. The paper compares a set of Byzantine protocols based on the typical model (nodes interconnected by a network) with our model. For this comparison to make sense we consider only randomized protocols, only with local random oracles (i.e., following Ben-Or [3]). This approach does not lead to particularly efficient protocols, but the purpose is to compare protocols that are as similar as possible. The conclusion from that comparison is that our approach manages to reduce the complexities of Byzantine protocols to complexities equivalent to fail-stop protocols, which are considerably better than those of previous similar Byzantine protocols.

The paper follows the line of research in systems extended with wormholes, but considers, for the first time, local (non-distributed), strictly asynchronous, randomized wormholes. This possibility of having local wormholes was envisaged when the concept was first introduced [24, 28] but has never been explored. The paper presents the first formalization of this type of model with I/O automata.

# References


[1] G. Avoine, F. Gartner, R. Guerraoui, and M. Vukolic. Gracefully degrading fair exchange with security modules. In *Proceedings of the Fifth European Dependable Computing Conference*, volume 3463 of *Lecture Notes in Computer Science*, pages 55–71. Springer-Verlag, April 2005.

[2] R. Baldoni, J. Helary, M. Raynal, and L. Tanguy. Consensus in Byzantine asynchronous systems. In *Proceedings of the International Colloquium on Structural Information and Communication Complexity*, pages 1–16, June 2000.

[3] M. Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. In *Proceedings of the 2nd ACM Symposium on Principles of Distributed Computing*, pages 27–30, August 1983.

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

[5] M. Castro and B. Liskov. A correctness proof for a practical Byzantine-fault-tolerant replication algorithm. Technical Report MIT/LCS/TM-590, MIT Laboratory for Computer Science, June 1999.

[6] B. Chor and C. Dwork. Randomization in Byzantine agreement. In *Advances in Computing Research 5: Randomness and Computation*, pages 443–497. JAI Press, 1989.

[7] M. Correia, N. F. Neves, L. C. Lung, and P. Veríssimo. Low complexity Byzantine-resilient consensus. *Distributed Computing*, 17(3):237–249, 2005.

[8] M. Correia, P. Veríssimo, and N. F. Neves. The design of a COTS real-time distributed security kernel. In *Proceedings of the Fourth European Dependable Computing Conference*, pages 234–252, October 2002.

[9] A. Doudou, B. Garbinato, and R. Guerraoui. Encapsulating failure detection: From crash-stop to Byzantine failures. In *International Conference on Reliable Software Technologies*, pages 24–50, May 2002.

[10] A. Doudou and A. Schiper. Muteness detectors for consensus with Byzantine processes. Technical Report 97/30, EPFL, 1997.

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

[12] P. Ezhilchelvan, A. Mostefaoui, and M. Raynal. Randomized multivalued consensus. In *Proceedings of the 4th IEEE International Symposium on Object-Oriented Real-Time Computing*, pages 195–200, May 2001.

[13] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[14] A. Frier, P. Karlton, and P. Kocher. The SSL 3.0 protocol. Netscape Communications Corp., November 1996.

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

[16] R. Guerraoui and A. Schiper. The generic consensus service. *IEEE Transactions on Software Engineering*, 27(1):29–41, January 2001.

[17] N. Itoi and P. Honeyman. Smartcard integration with Kerberos v5. In *Proceedings of the USENIX Workshop on Smartcard Technology*, May 1999.

[18] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, San Mateo, CA, 1996.

[19] N. Lynch and M. Tuttle. An Introduction to Input/Output automata. *CWI-Quarterly*, 2(3):219–246, Sep 1989.

[20] D. Malkhi and M. Reiter. Unreliable intrusion detection in distributed computations. In *Proceedings of the 10th Computer Security Foundations Workshop*, pages 116–124, June 1997.

[21] F. Meyer and D. Pradhan. Consensus with dual failure modes. In *Proceedings of the 17th IEEE International Symposium on Fault-Tolerant Computing*, pages 214–222, July 1987.

[22] A. Mostefaoui, M. Raynal, and F. Tronel. From binary consensus to multivalued consensus in asynchronous message-passing systems. *Information Processing Letters*, (73):207–212, 2000.

[23] N. F. Neves, M. Correia, and P. Veríssimo. Solving vector consensus with a wormhole. *IEEE Transactions on Parallel and Distributed Systems*, 2005. Accepted for publication.

[24] D. Powell and R. J. Stroud, editors. *MAFTIA: Conceptual Model and Architecture. Project MAFTIA deliverable D2*. November 2001. http://www.research.ec.org/maftia/deliverables/D2fin.pdf.

[25] M. O. Rabin. Randomized Byzantine generals. In *Proceedings of the 24th Annual IEEE Symposium on Foundations of Computer Science*, pages 403–409, November 1983.

[26] V. Shoup and A. D. Rubin. Session key distribution using smart cards. In Springer-Verlag, editor, *Proceedings of the International Conference on the Theory and Application of Cryptographic Techniques, Eurocrypt'96*, volume 1070 of *Lecture Notes in Computer Science*, May 1996.

[27] J. D. Tygar and B. S. Yee. Dyad: A system for using physically secure coprocessors. In *Proceedings of the Joint Harvard-MIT Workshop on Technological Strategies for the Protection of Intellectual Property in the Network Multimedia Environment*, April 1993.

[28] P. Veríssimo. Uncertainty and predictability: Can they be reconciled? In *Future Directions in Distributed Computing*, volume 2584 of *Lecture Notes in Computer Science*, pages 108–113. Springer-Verlag, 2003.

[29] P. Veríssimo, A. Casimiro, and C. Fetzer. The Timely Computing Base: Timely actions in the presence of uncertain timeliness. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 533–542, June 2000.

[30] P. Veríssimo, N. F. Neves, and M. Correia. Intrusion-tolerant architectures: Concepts and design. In R. Lemos, C. Gacek, and A. Romanovsky, editors, *Architecting Dependable Systems*, volume 2677 of *Lecture Notes in Computer Science*, pages 3–36. Springer-Verlag, 2003.

# A Fail-stop $\lfloor (n-1)/3 \rfloor$-Resilient Binary Consensus Protocol

This section presents a straightforward modification of Bracha's Byzantine-resilient binary consensus protocol [4] to tolerate $\lfloor (n-1)/3 \rfloor$ crash faults. This protocol is used in the evaluation of our Byzantine-resilient consensus protocols (see Section 4). The modification is essentially the removal of the 'reliable broadcast' primitive and the 'correctness enforcement' scheme used in [4] to constrain the behavior of malicious processes. We also generalize Bracha's assumption of $n = 3f + 1$ to $f = \lfloor (n-1)/3 \rfloor$.

The protocol is presented in Algorithm 4 following the original format. Also following the original presentation, the algorithm does not terminate when a decision is made. This can be done by making the processes that decided broadcast a halting message.

**Algorithm 4** Fail-stop $\lfloor (n-1)/3 \rfloor$-Resilient Binary Consensus Protocol

*$i_p$ is set to the value proposed by the process before the first round.*
*$(d, v)$ is a special value that is used to try to decide $v$.*

**Round($k$)**: (by process $p$)

1. *Broadcast($i_p$)* and wait for $n - f$ messages.
   $i_p :=$ majority value of the messages.

2. *Broadcast($i_p$)* and wait for $n - f$ messages.
   If more than $n/2$ of the messages have the same value, then $i_p := (d, v)$.

3. *Broadcast($i_p$)* and wait for $n - f$ messages.
   If there are at least $(n - f)$ $(d, v)$ messages then *Decide* $v$.
   If there are at least $(n - 2f)$ $(d, v)$ messages then $i_p := v$.
   Otherwise, $i_p := 1$ or $0$ with probability $1/2$.
   Go to step 1 of round $k + 1$.

**Theorem 3** *If at most $f = \lfloor \frac{n-1}{3} \rfloor$ processes are stopped, then the protocol presented in Algorithm 4 solves consensus as specified by properties Validity-1, Agreement and Termination.*

*Proof (sketch):*

*Validity-1.* The protocol is binary, therefore only two values can be proposed. The property would be false only if all processes proposed 0 (resp. 1) and a correct process decided 1 (resp. 0). A simple inspection of the protocol shows this is impossible: if all processes propose the same value then all decide it.

*Agreement.* Two processes cannot decide different values (0 and 1) in the same round since they would need respectively $n - f$ $(d, 0)$ messages and $n - f$ $(d, 1)$ messages. This is clearly impossible since each process can only send one message per round and step, and $n - f + n - f > n$.

Now, without loss of generality assume process $p_0$ decides 0 in round $k$ and process $p_1$ decides 1 in round $k' > k$. Process $p_0$ must have received $n - f$ $(d, 0)$ messages in step 3 of round $k$ so all other processes received at most $f$ $(d, 1)$ messages in the same round/step. Therefore, all other processes set their variable $i$ to $v$ since all received at least $n - 2f$ $(d, 0)$ messages (step 3). An inspection of the protocol shows that if all processes set $i$ to $v$ in round $k$ then in round $k + 1$ process $p_1$ decides 0. A contradiction.

*Termination.* An inspection shows that the protocol cannot deadlock. We just proved that if a process decides $v$ then all correct processes decide $v$ not after the next round. If no process decides, there is an increasing probability that eventually $n - f$ processes set $i$ to the same value $v$ in step 3 (say, in round $k$). When this happens, all processes receive at least $n - 2f$ messages with $v$ in step 1 of round $k + 1$ (since only $f$ processes may broadcast a different value) and set $i$ to $v$. In step 2, all processes broadcast $v$ and set $i$ to $(d, v)$. Finally, in step 3 all decide. □