The Computer Journal Advance Access published November 16, 2005

doi:10.1093/comjnl/bxh145

From Consensus to Atomic Broadcast: Time-Free Byzantine-Resistant Protocols without Signatures

MIGUEL CORREIA*, NUNO FERREIRA NEVES AND PAULO VERÍSSIMO

Faculdade de Ciências da Universidade de Lisboa, Campo Grande, 1749-016 Lisboa, Portugal
**Corresponding author: mpc@di.fc.ul.pt*

This paper proposes a stack of three Byzantine-resistant protocols aimed to be used in practical distributed systems: multi-valued consensus, vector consensus and atomic broadcast. These protocols are designed as successive transformations from one to another. The first protocol, multi-valued consensus, is implemented on top of a randomized binary consensus and a reliable broadcast protocol. The protocols share a set of important structural properties. First, they do not use digital signatures constructed with public-key cryptography, a well-known performance bottleneck in this kind of protocols. Second, they are time-free, i.e. they make no synchrony assumptions, since these assumptions are often vulnerable to subtle but effective attacks. Third, they are completely decentralized, thus avoiding the cost of detecting corrupt leaders. Fourth, they have optimal resilience, i.e. they tolerate the failure of $f = \lfloor (n-1)/3 \rfloor$ out of a total of n processes. In terms of time complexity, the multi-valued consensus protocol terminates in a constant expected number of rounds, while the vector consensus and atomic broadcast protocols have $O(f)$ complexity. The paper also proves the equivalence between multi-valued consensus and atomic broadcast in the Byzantine failure model without signatures. A similar proof is given for the equivalence between multi-valued consensus and vector consensus. These two results have theoretical relevance since they show once more that consensus is a fundamental problem in distributed systems.

Received 16 November 2004; revised 13 October 2005

1. INTRODUCTION

Distributed protocols capable of tolerating Byzantine faults have been studied for more than two decades [1, 2, 3, 4]. Recently, interest in these protocols has gained a new momentum under the designation of *intrusion tolerance* [5]. The basic idea is that the security concepts of attack, intrusion and vulnerability can be considered as *faults*, more precisely as arbitrary faults, also called Byzantine faults. A consequence of this assertion is that Byzantine-resistant protocols can be important building blocks for the construction of secure systems.

Byzantine-resistant (or intrusion-tolerant) protocols usually have higher time and message complexities than crash-tolerant protocols do. They are also more CPU-time demanding since they must use cryptography,[1] and often public-key cryptography. This CPU-time issue is frequently dismissed since the processing power of computers is constantly increasing. However, new classes of computing environments are appearing in which resources are scarce, e.g. embedded systems. This is an important motivation for the design of less CPU-time consuming intrusion-tolerant protocols. Moreover, public-key cryptography operations can be an important bottleneck for the performance of intrusion-tolerant systems even in more powerful hardware. Castro and Liskov designed an intrusion-tolerant NFS system which performs on average only 3% slower than standard NFS, in part due to avoiding the use of signatures based on public-key cryptography [6].

An argument of this paper is that the design of efficient Byzantine-resistant protocols is crucial for the implementation of practical intrusion-tolerant systems; therefore these protocols have to avoid as much as possible the use of public-key cryptography. Moreover, practical intrusion-tolerant systems require protocols with other characteristics, like strict asynchrony, optimal resilience and low time complexity. The paper provides a modular and consistent family of protocols with these properties.

Paper results. The paper presents a stack of three message-passing Byzantine-resistant protocols: multi-valued

[1] Here we are talking about practical systems. Theoretically we can assume private channels connecting the processes, therefore cryptography is not an absolute requirement.

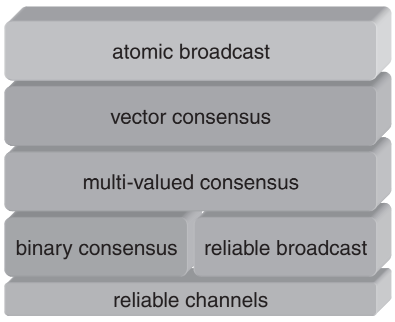

FIGURE 1. Protocol architecture.

consensus, vector consensus and atomic broadcast (see Figure 1). Consensus is a distributed systems problem with both theoretical and practical interest. The problem can be stated this way: how does a set of distributed processes achieve agreement on a value despite a number of process failures? The paper implements two flavors of consensus: *multi-valued consensus* that makes agreement on values with an arbitrary size and *vector consensus* that makes agreement on a vector with the values proposed by several of the processes. An *atomic broadcast* protocol is a communication protocol that delivers the same messages to all processes in the same order. Atomic broadcast is, for instance, the main component of fault-tolerant systems based on the state-machine approach, with both crash [7] and Byzantine faults [6, 8]. The protocols in the paper do not solve consensus from scratch but are built on top of a randomized binary consensus protocol (e.g. [9, 10]) and a reliable broadcast protocol (e.g. [9])—see Figure 1.

The problem of consensus has been studied with different system models, such as the synchronous and the asynchronous time models, the crash and the arbitrary failure models, and in message-passing and shared-memory systems. In asynchronous systems, consensus has been shown to be constrained by the FLP impossibility result, which says that it is impossible to solve consensus deterministically in a completely asynchronous system [11]. Consequently, various researchers have proposed ways to circumvent this limitation:[2] using randomization [3, 4, 9, 10, 12, 13, 14], making synchrony or timing assumptions on the behavior of the system [15, 16, 17], using failure detectors [18, 19, 20, 21, 22] or ordering oracles [23], using wormholes [24, 25, 26] or imposing conditions on inputs [27, 28]. Some common misunderstandings about consensus and FLP are discussed in [29].

The protocols presented in the paper are intended to be practical. Their modularity allows a system designer to implement only the protocols he/she needs, instead of the full stack. Moreover, the protocols share the following set of important structural properties:

- *Signature free*. The protocols do not use signatures based on public-key cryptography.
- *Asynchrony*. The protocols are asynchronous, i.e. there are no synchrony assumptions whatsoever.
- *Decentralization*. Decisions are taken in a decentralized way, i.e. there are no coordinators, leaders or token-holders.
- *Optimal resilience*. The protocols tolerate $f = \lfloor (n-1)/3 \rfloor$ faulty processes out of a total of n processes.

A stack of protocols with this combination of characteristics is novel, to the best of our knowledge. We argue that all of them are important if the protocols are to be used in practice. The argument for avoiding public-key cryptography (first property) has already been done above, so let us discuss the importance of the other three properties.

Many protocols in the literature are designated 'asynchronous' but make synchrony assumptions, either explicitly [15, 16, 17] or contained in the unreliable failure detector abstraction [19, 20, 21, 22]. These assumptions can make the protocols vulnerable to subtle but effective attacks in the domain of time, something that cannot happen in time-free systems. Some discussion about these kinds of attacks and the corresponding vulnerabilities can be found in [6, 14]. Our protocols are time-free or strictly asynchronous (second property) but circumvent FLP by being built on top of a randomized binary consensus protocol. Randomized protocols have a probability of satisfying their properties that increase with the number of rounds executed. The protocols in the paper satisfy deterministically all their properties except termination; that nevertheless happens with probability 1.

The third property—decentralization—is important because it eludes the need for detecting faulty coordinators, leaders or token-holders. This detection usually has a price in terms of time and messages transmitted. Moreover, even a common failure like a process crash cannot be detected in a strictly asynchronous system, since there are no bounds on the communication delays.

The *resilience* of a protocol can be defined as the maximum number of faults in the presence of which the protocol still behaves according to its specification. The optimal resilience for asynchronous consensus has been shown to be $\lfloor (n-1)/3 \rfloor$ [13] and we prove that atomic broadcast is an equivalent problem, so the optimal resilience is the same (this has already been claimed by [13, 30, 31]). Optimal resilience is an important property because the need for additional processes to tolerate the same number of faults involves a cost in terms of additional resources (e.g. additional hardware).

The evaluation of a distributed protocol is usually made in terms of time and message complexities, so we evaluate the protocols in terms of both. In asynchronous systems, time complexity is usually measured in terms of maximum number of *asynchronous rounds*. An asynchronous round involves a process sending a message and receiving one or more messages sent by the other processes. For randomized protocols, the metric is usually the *expected number of asynchronous rounds*. Our multi-valued consensus protocol has time complexity $O(1)$, i.e. it has a constant expected number of rounds. The complexities of the vector consensus and the atomic broadcast protocols are both $O(f)$, although

[2]We use the expression *to circumvent FLP* since it is common in the literature. However, what the expression means is that the model for which FLP was stated is modified so that FLP no longer applies.

they are reduced to $O(1)$ when all processes are correct. These complexities are at least as good as previous works, except for one vector consensus that manages to have time complexity $O(1)$ at the cost of a significatively higher message complexity [32]. The message complexities, measured in *expected number of messages sent*, are usually higher than those obtained by protocols that use signatures, so there is a tradeoff involved.

The paper has a further contribution. Atomic broadcast has been shown to be equivalent to multi-valued consensus in systems prone to crash faults [18, 30]. For systems prone to Byzantine faults with signatures there is also a proof [31]. Here we prove this equivalence without the requirement of signatures. Moreover, we also prove that multi-valued consensus and vector consensus are equivalent in the same system model.

Paper organization. The paper is organized as follows. The following section defines the system model and the two components used by our protocols: reliable broadcast and binary consensus. Section 3 presents our multi-valued consensus protocol and proves its correctness. Sections 4 and 5 present respectively, the vector consensus and atomic broadcast protocols. Section 6 proves the equivalence multi-valued consensus/atomic broadcast, and Section 7 proves the equivalence multi-valued consensus/vector consensus. Section 8 assesses the performance of the protocols. Section 9 discusses some related work and Section 10 concludes the paper.

2. DEFINITIONS

2.1. System model

The system is composed of a set of n processes $P = \{p_1, p_2, \ldots, p_n\}$. 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. We assume that at most $f = \lfloor (n-1)/3 \rfloor$ processes can fail and we call these processes *corrupt*. These failures can be Byzantine, meaning that processes can stop, omit messages, send incorrect messages, send several messages with the same identifier etc. Additionally, corrupt processes can pursue their goal of breaking the properties of the protocol alone or in collusion with other corrupt processes.

Processes are fully-connected by *reliable channels* with two properties: if the sender and the recipient of a message are both correct then (i) the message is eventually received and (ii) the message is not modified in the channel.[3]

The system is asynchronous, which means that there are no bounds on the processing times or communication delays.

[3]In practice, reliable channels have to be implemented using retransmissions and cryptography, e.g. with message authentication codes (MACs) that are based on symmetric cryptography [33]. Processes have to share symmetric keys in order to use MACs. In the paper we assume these keys are distributed before the protocol is executed. In practice, this can be solved using key distribution protocols available in the literature, but the issue is out of the scope of the paper.

2.2. Reliable broadcast

A reliable broadcast protocol ensures essentially that all correct processes deliver the same messages, and that messages broadcast by correct processes are delivered. Moreover, it ensures that no different messages with the same identifier are delivered. This identifier includes the typical information in a protocol header: protocol type, sender, broadcast channel and sequence number. An example of an asynchronous Byzantine-resistant reliable broadcast protocol is the one proposed by Bracha [9]. We consider that the reliable broadcast is executed by calling the function `R_Broadcast`(M) (see, e.g. Algorithm 1 below).

Formally, a reliable broadcast protocol can be defined in terms of the following properties [30, 31]:

- *RB1 Validity:* if a correct process broadcasts a message M, then some correct process eventually delivers M.
- *RB2 Agreement:* if a correct process delivers a message M, then all correct processes eventually deliver M.
- *RB3 Integrity:* for any identifier *ID*, every correct process p delivers at most one message M with identifier *ID*, and if $sender(M)$ is correct then M was previously broadcast by $sender(M)$.

The predicate $sender(M)$ gives the field of the message header that identifies its sender. We consider that the sender also delivers the messages it broadcasts.

Note that property RB3 prevents the behavior we discussed above: it prevents a correct process from delivering two messages with the same *ID* broadcast by the same malicious process. This is important for the protocols in this paper, as we will see later. However, it has only to be satisfied during the execution of the protocol that uses reliable broadcast, not forever.

2.3. Binary consensus

A binary consensus protocol performs consensus on a binary value $b \in \{0, 1\}$. The problem can be formally defined in terms of three properties:

- *BC1 Validity:* if all correct processes propose the same value b, then any correct process that decides, decides b.
- *BC2 Agreement:* no two correct processes decide differently.
- *BC3 Termination:* every correct process eventually decides.

This definition has two immediate consequences that we state and prove for later reference in the paper.

THEOREM 1. *If a correct process decides b, then b was proposed by some process.*

Proof. If all processes propose the same value b, then BC1 guarantees that this is the value decided. If processes propose different values then the value decided must have been proposed since there are only two possible values: $\{0, 1\}$. □

THEOREM 2. *If a value b is proposed only by corrupt processes, then no correct process that decides, decides b.*

ALGORITHM 1. Multi-valued consensus protocol (for process p_i).

```
Function M_V_Consensus ($v_i$, cid)
Initialization:
 1: INIT_delivered$_i$ ← ∅;          {INIT messages delivered}
 2: activate task (T1,T2);
Task T1:
 3: R_Broadcast ( ⟨INIT, $v_i$, cid, i⟩ );
 4: wait until (at least $(n - f)$ INIT messages have been delivered);
 5: $\forall_j$: if (⟨INIT, $v_j$, cid, j⟩ has been delivered) then $V_i[j] \leftarrow v_j$; else $V_i[j] \leftarrow \perp$;
 6: if $(\exists^1_v : \#_v(V_i) \geq (n - 2f))$ then
 7:     $w_i \leftarrow v$;
 8: else
 9:     $w_i \leftarrow \perp$;
10: R_Broadcast ( ⟨VECT, $w_i$, $V_i$, cid, i⟩ );
11: wait until (at least $(n - f)$ valid messages ⟨VECT, $w_j$, $V_j$, cid, j⟩ have been delivered);
12: $\forall_j$: if (⟨VECT, $w_j$, $V_j$, cid, j⟩ has been delivered) then $W_i[j] \leftarrow w_j$; else $W_i[j] \leftarrow \perp$;
13: if $(\forall_{j,k}\ W_i[j] \neq W_i[k] \Rightarrow W_i[j] = \perp$ or $W_i[k] = \perp)$ and $(\exists_w: \#_w(W_i) \geq (n - 2f))$ then
14:     $b_i \leftarrow 1$;
15: else
16:     $b_i \leftarrow 0$;
17: $c_i \leftarrow$ B_Consensus($b_i$, cid);
18: if $(c_i = 0)$ then
19:     return $\perp$;
20: wait until (at least $(n - 2f)$ valid messages ⟨VECT, $v_j$, $V_j$, cid, j⟩ with $v_j = v$ have been delivered);
21: return v;
Task T2:
22: when $m_i$ = ⟨INIT, $v_j$, cid, j⟩ is delivered do
23:     INIT_delivered$_i$ ← INIT_delivered$_i$ $\bigcup \{m_i\}$;
```

Proof. If a value b is proposed only by corrupt processes then all correct processes proposed $\neg b$ since $b \in \{0, 1\}$. Therefore, BC1 guarantees that any correct process that decides, decides $\neg b$, i.e. does not decide b. □

Besides satisfying this definition, the binary consensus protocol to be used in the stack has to be compatible with the structural properties given in the introduction: it cannot use public-key signatures, has to be asynchronous, has to take decisions in a decentralized way and has to have optimal resilience. Examples of protocols that satisfy these requirements are [9, 10]. Appendix A presents an efficient protocol that also satisfies these requirements, although it does not avoid public-key cryptography entirely (it uses a variation of the Diffie–Hellman problem).

Throughout the paper we consider that the binary consensus protocol is executed by calling the function `B_Consensus`(b, $bcid$), where b is the binary value proposed and $bcid$ the protocol execution identifier.

3. MULTI-VALUED CONSENSUS

The first protocol of the stack proposed in the paper is a multi-valued consensus. The definition of the problem is similar to the binary consensus, except that processes can propose values with arbitrary length $v \in \mathcal{V}$ ($\mathcal{V}$ is the domain of values that can be proposed). The protocol can decide one of the proposed values or a default value $\perp \notin \mathcal{V}$. The definition is:

- *MVC1 Validity 1.* If all correct processes propose the same value v, then any correct process that decides, decides v.
- *MVC2 Validity 2.* If a correct process decides v, then v was proposed by some process or $v = \perp$.
- *MVC3 Validity 3.* If a value v is proposed only by corrupt processes, then no correct process that decides, decides v.
- *MVC4 Agreement.* No two correct processes decide differently.
- *MVC5 Termination.* Every correct process eventually decides.

The problem of multi-valued consensus is often stated in terms of the properties MVC4, MVC5 and either MVC1 or MVC2 (e.g. MVC1 in [15, 34, 35] and MVC2 in [20, 22, 21]). We define consensus using all three validity properties following the definition used in the original Byzantine Generals paper [2].[4] Moreover, a consensus protocol that satisfies only MVC1 or MVC2 has limited interest in practice. Property MVC1 does not say anything about which value is decided when the correct processes do not propose the same value. Property MVC2 does not impose that the value decided is proposed by a correct process. Notice that we proved, respectively in Theorems 1 and 2, that for binary consensus, Validity 1 implies Validity 2 and Validity 3.

A word is due about why the other papers do not use a definition more similar to ours. The reason is probably that the interest of these other papers in consensus is theoretical. These papers are mostly interested in proving that consensus can be solved under a certain model, e.g. in the presence of partial synchrony [15], with a *quietness* failure detector [34] or with a *muteness* failure detector [20]. Our interest on Byzantine consensus, on the contrary, follows, for example, Guerraoui and Schiper that aim to solve practical problems using this type of protocol, albeit with a crash failure model in their case [36].

3.1. The protocol

The protocol is presented in Algorithm 1. Local variables are designated by lowercase letters with a subscript indicating the process to which they belong: w_i, b_i, c_i in process p_i. Vectors have one entry per process in P and are designated

[4]The original definition is in the context of the 'Byzantine Generals' metaphor used in the paper: '(1) All loyal generals decide upon the same plan of action; (2) A small number of traitors cannot cause the loyal generals to adopt a bad plan.' [2].

by an uppercase letter, e.g. vector V_i has entries $V_i[1]$, $V_i[2], \ldots, V_i[n]$. Function $\#_x(V)$ counts the number of occurrences of x in vector V. The maximum number of faulty processes is a function of the total number of processes n: $f = \lfloor (n-1)/3 \rfloor$. The protocol uses two types of messages: INIT and VECT. The content of messages is represented inside angles: $\langle \ldots \rangle$. A set called INIT_delivered$_i$ is used to store the received INIT messages. A call to *return* causes the termination of all the protocol's tasks. The value returned is the result of the protocol, i.e. the decided value.

Function `M_V_Consensus` is called with two arguments: the value proposed by the process (v_i) and the consensus identifier (cid). There is an initialization and tasks T1 and T2 are started concurrently (lines 1 and 2). Task T1 does most of the work, while task T2 simply receives INIT messages and stores them in INIT_delivered$_i$ (lines 22 and 23).

Task T1 begins by reliably broadcasting an INIT message with the value v_i proposed by process p_i (line 3). The identifier of the message includes the message type (INIT), the consensus (cid) and sender identifiers (i). Then, the task waits for the reception of $(n-f)$ INIT messages (including its own) and stores the proposed values in vector V_i (lines 4 and 5). The reliable broadcast protocol guarantees that two correct processes p_i and p_j do not receive different proposals from the same process (see Section 2.2). However, V_i can be different from V_j since the first $(n-f)$ INIT messages received by the two processes do not have to be the same.

If all correct processes propose the same value v then all correct processes receive at least $(n-2f)$ INIT messages with v. If a process receives this number of messages with a value v, then it selects this value (lines 6 and 7) and reliably broadcasts it to all processes together with the vector V_i that justifies the selection (line 10). Otherwise, it selects the default value $\perp$, which it also broadcasts. After broadcasting this message (VECT), the process waits for $(n-f)$ *valid* VECT messages, i.e. messages known to have a vector with real proposals and a value substantiated by those proposals. The identifier of a message VECT includes the protocol type (VECT) and also the consensus (cid) and sender identifiers (i).

DEFINITION 1. *A message $\langle$VECT, w_j, V_j, cid, j$\rangle$ is said to be valid at process p_i iff:*

- *$\forall_k$, $V_j[k] = \perp$ or there is a message $\langle$INIT, v_k, cid, k$\rangle \in$ INIT_delivered$_i$ so that $V_j[k] = v_k$*
- *$w_j \neq \perp \Leftrightarrow \#_{w_j}(V_j) \geq (n-2f)$*

If the process does not receive two VECT messages with different values $w \neq w'$, and it receives at least $(n-2f)$ messages with w, it proposes 1 for the binary consensus, otherwise it proposes 0 (lines 13–16). If the binary consensus decides 0, the vector consensus protocol decides on the default value $\perp$ (lines 17–19).

If the binary consensus decides 1, the process waits until it received $(n-2f)$ valid VECT messages with the same value v (line 20). The process does not wait until it received $(n-2f)$ valid VECT messages with the same value *in line 20* but rather until it received cumulatively these messages since the beginning of the protocol execution (some of them were received in line 11). When these messages are received, the protocol returns v (line 21). The protocol can be sure that there can only be one value v for which a correct process can consider $(n-2f)$ VECT messages to be valid, or two different correct processes might decide different values. We show that this is true in the proof of Theorem 6.

3.2. Correctness proof

The protocol in Algorithm 1 is correct if it satisfies properties MVC1–MVC5. A preliminary result is given by the following lemma:

LEMMA 1. *If a message $\langle$VECT, w_i, V_i, cid, i$\rangle$ is reliably broadcast by a correct process p_i, then eventually all correct processes will consider it valid.*

Proof. The INIT messages are reliably broadcast (line 3). Consequently, all correct processes eventually deliver the same INIT messages (properties RB1–RB3 in Section 2.2). A correct process only puts in V_i values v_j it received in INIT messages (line 5). Therefore, for every value v in a VECT message sent by a correct process, there is an INIT message that is eventually delivered by all correct processes. Additionally, a correct process always sends VECT messages with at least $(n-f)$ values (lines 4, 5 and 10). This proves the lemma, attending to the definition of *valid* message. □

THEOREM 3. (Validity 1). *If all correct processes propose the same value v, then any correct process that decides, decides v.*

Proof. If all correct processes propose the same value v, then all processes deliver at least $(n-2f)$ INIT messages with v and at most f INIT messages with $v' \neq v$ (at most f processes are corrupt). Consequently, all correct processes make $w_i = v$, and send this value in a VECT message (lines 6–10). Moreover, all correct processes deliver at least $(n-2f)$ valid VECT messages in line 11 (Lemma 1). No valid VECT message can have $w_i \neq v$ since at most f (corrupt) processes send INIT messages with a value different from v. Therefore, all correct processes make $b_i = 1$ (lines 13 and 14). All correct processes start a binary consensus protocol (line 17) that decides 1 (property BC1). The value decided is necessarily v (lines 18–21). □

THEOREM 4. (Validity 2). *If a correct process decides v, then v was proposed by some process or $v = \perp$.*

Proof. The proof is obtained with a trivial inspection of the protocol. □

THEOREM 5. (Validity 3). *If a value v is proposed only by corrupt processes, then no correct process that decides, decides v.*

Proof. The proof is by contradiction. If a correct process decides v then it received at least $(n-2f)$ valid VECT messages with v. For a VECT message to be valid there has to be at least $(n-2f) > f$ INIT messages with v, but the theorem assumes only corrupt processes proposed v: a contradiction. □

THEOREM 6. (Agreement). *No two correct processes decide differently.*

Proof. All correct processes get the same decision from the binary consensus protocol (property BC2). The proof can be divided into two cases, depending on the value c_i decided by the binary consensus (line 17). The first case, $c_i = 0$, its trivial: all correct processes decide $\perp$ (lines 18 and 19).

For the second case, $c_i = 1$, the proof is by contradiction. Two correct processes p_1 and p_2 decide differently if: (i) p_1 delivers $(n - 2f)$ valid VECT messages with the same value v_1 (line 20); and (ii) p_2 delivers also $(n - 2f)$ valid VECT messages but with a value $v_2 \neq v_1$.

The binary consensus protocol decided 1, so at least one correct process p_1 (without loss of generality) proposed 1 in line 17 (Theorem 2). p_1 proposed 1, therefore the two conditions in line 13 were satisfied. The second condition implies that p_1 received at least $(n - 2f)$ valid VECT messages with value v_1 in line 11. The first condition implies that p_1 did not receive any valid VECT message with a value different from v_1. Therefore, p_1 received:

- m_1 valid VECT messages with v_1, and $m_1 \geq (n - 2f)$;
- m_2 valid VECT messages with $\perp$, and $m_1 + m_2 \geq (n - f)$.

Now, the proof assumes p_2 received $m_3 = (n - 2f)$ valid VECT messages with v_2. However, there can be at most one valid VECT message per process for an execution of the consensus protocol, totalizing n, due to the reliable broadcast protocol's property RB3. Therefore, we have:

$$\begin{aligned} &m_1 + m_2 + m_3 \leq n \\ &\quad \Rightarrow (n - f) + (n - 2f) \leq n \\ &\quad \Leftrightarrow n \leq 3f \end{aligned}$$

This is a contradiction since we assume that $f = \lfloor (n-1)/3 \rfloor$, what implies that $n > 3f$. □

THEOREM 7. (Termination). *Every correct process eventually decides.*

Proof. Correct processes decide when they execute lines 19 or 21. The places of the protocol in which we have to prove that the protocol makes progress are the two executions of the reliable broadcast protocol (lines 3 and 4 and 10 and 11), the execution of the binary consensus protocol (line 17) and the reception of VECT messages in line 20.

The termination of the reliable broadcast protocol is guaranteed by its Validity and Agreement properties (RB1, RB2). All correct processes eventually deliver $(n - f)$ INIT messages in line 4 because all correct processes reliably broadcast an INIT message in line 3, and there are at most f corrupt processes. This proves that the protocol makes progress in lines 3 and 4. The justification for lines 10 and 11 is identical. The binary consensus protocol executed in line 17 is guaranteed to terminate by property BC3.

The protocol waits for the condition in line 20 only if the binary consensus decides 1. If all correct processes had proposed 0 for the binary consensus, then the process would have decided 0 (lines 17–19). Therefore, at least one correct process proposed 1 for the binary consensus. A correct process proposes 1 for the binary consensus only if it delivered $(n - 2f)$ valid VECT messages with the same value w (second condition in line 13 and lines 11 and 12). The VECT messages are reliably broadcast, therefore if a correct process delivers $(n-2f)$ valid VECT messages with w, then all correct processes eventually do the same. Therefore no correct process blocks in line 20 and all terminate. □

4. VECTOR CONSENSUS

Vector consensus makes agreement on a vector with a subset of the values proposed, instead of a single value [20, 26]. In systems where Byzantine faults can occur, the vector is useful, e.g. to implement atomic broadcast, only if a majority of its values were proposed by correct processes. Therefore, the decided vector needs to have at least $(2f+1)$ values. This problem is ultimately an adaptation for asynchronous systems of the classical problem of *interactive consistency* defined for synchronous systems [1]. The difference between the two problems is that interactive consistency makes agreement on a vector with the values proposed by all correct processes, while vector consensus guarantees only that the majority of the values were proposed by correct processes. The reason for this difference is that in asynchronous systems it is not possible to ensure that the vector has the proposals of all correct processes, since they can be arbitrarily delayed.

Vector consensus can be defined in terms of the following properties:

- *VC1 Vector validity:* every correct process that decides, decides on a vector V of size n:
  - $\forall_{p_i}$: if p_i is correct, then either $V[i]$ is the value proposed by p_i or $\perp$;
  - at least $(f + 1)$ elements of V were proposed by correct processes.
- *VC2 Agreement:* no two correct processes decide differently.
- *VC3 Termination:* every correct process eventually decides.

4.1. The protocol

The protocol is implemented by the function `Vector_Consensus` presented in Algorithm 2. The arguments are the value proposed (v_i) and the vector consensus identifier (vcid). The protocol starts by reliably broadcasting a VC_INIT message with the value proposed by the process (line 2). This message is identified by the protocol type (VC_INIT), the vcid and the sender (i). Then, the protocol runs one or more rounds until a decision is made (lines 3–8).

The algorithm begins each round by waiting for the reception of $(n - f + r_i)$ VC_INIT messages (line 4). Notice that line 4 does not restart from scratch waiting for the $(n - f + r_i)$ messages but rather waits until that number of messages has cumulatively been received since the beginning

ALGORITHM 2. Vector consensus protocol (for process p_i).

```
Function Vector_Consensus ($v_i$, vcid)
1: $r_i \leftarrow 0$;                                   {round number}
2: R_Broadcast ( ⟨VC_INIT, $v_i$, vcid, i⟩ );
3: repeat
4:     wait until (at least $(n - f + r_i)$ VC_INIT messages
       have been delivered);
5:     $\forall_j$: if ( ⟨VC_INIT, $v_j$, vcid, j⟩ has been delivered)
       then $W_i[j] \leftarrow v_j$; else $W_i[j] \leftarrow \perp$;
6:     $V_i \leftarrow$ M_V_Consensus ($W_i$, (vcid,$r_i$));
7:     $r_i \leftarrow r_i + 1$;
8: until ($V_i \neq \perp$);
9: return $V_i$;
```

of the execution of the protocol. Next, the process builds a vector W_i with the values it received from other processes (at least $(n - f)$ in round 0, $(n - f + 1)$ in round 1, ...) and proposes the vector for a multi-valued consensus (lines 5 and 6). The identifier of the multi-valued consensus is unique for each execution by using a combination of vcid and the round number, r_i.

VC_INIT is reliably broadcast, therefore all correct processes will eventually receive the same VC_INIT messages and build identical W vectors. When enough processes propose the same W vector for the multi-valued consensus, W is decided by this protocol and immediately after by the vector consensus (lines 6–9).

4.2. Correctness proof

The protocol in Algorithm 2 is correct if it satisfies the properties VC1, VC2 and VC3.

Theorem 8. (Vector validity). *Every correct process that decides, decides on a vector V of size n: (i) $\forall_{p_i}$: if p_i is correct, then either V[i] is the value proposed by p_i or $\perp$; and (ii) at least $(f+1)$ elements of V were proposed by correct processes.*

Proof. The values proposed by each process are reliably broadcast so all correct processes eventually deliver the same values (lines 2 and 4). Any correct process calls M_V_Consensus in line 6 with a vector W_i that satisfies the two conditions of the theorem: (i) each entry j of the vector contains either the value proposed by process p_j or $\perp$; and (ii) W_i has at least $(n - f)$ elements from which at least $(n - 2f) \geq (f + 1)$ were proposed by correct processes (at most f processes are corrupt). $(n - 2f)$ must be greater or equal to $(f + 1)$ because $f = \lfloor (n-1)/3 \rfloor$. The value decided by the protocol (line 9) is the value decided on the last execution of the multi-valued consensus (line 6). This value is one of the values proposed (property MVC2) and cannot have been proposed only by corrupt processes (property MVC3). Therefore, the value must have been proposed by at least one correct process so the two conditions of the theorem are satisfied. □

Theorem 9. (Agreement). *No two correct processes decide differently.*

Proof. The value decided is equal to the value decided on the last execution of the multi-valued consensus (lines 5 and 6). All correct processes execute the same sequence of multi-vector consensuses because the identifier of each execution includes the round number (line 6). Therefore, the theorem is a trivial consequence of the Agreement property MVC4 of the multi-valued consensus. □

Theorem 10. (Termination). *Every correct process eventually decides.*

Proof. All VC_INIT messages reliably broadcast by correct processes are eventually delivered by all correct processes (properties RB1–RB3). Let p_i be any correct process. Process p_i executes one or more calls to M_V_Consensus, and each of these calls eventually terminates (property MVC5). Each round of the loop, p_i waits for one more VC_INIT message (line 4) before engaging in the multi-valued consensus (line 6). If p_i does not leave the loop and terminates before, the latest by round $r = f$ process p_i and all other correct processes propose for the multi-valued consensus a vector with the values from all processes. Therefore, in that round all correct processes propose the same vector, the multi-valued consensus decides a value different from $\perp$ (property MVC1) and the protocol terminates (lines 8 and 9). □

5. ATOMIC BROADCAST

The problem of atomic broadcast, or total order reliable broadcast, is the problem of delivering the same messages in the same order to all processes. The definition of the problem is equal to the definition of reliable broadcast plus a total order property:

- *AB1 Validity:* if a correct process broadcasts a message M, then some correct process eventually delivers M.
- *AB2 Agreement:* if a correct process delivers a message M, then all correct processes eventually deliver M.
- *AB3 Integrity:* for any identifier *ID*, every correct process p delivers at most one message M with identifier *ID*, and if *sender*(M) is correct then M was previously broadcast by *sender*(M).
- *AB4 Total order:* if two correct processes deliver two messages M_1 and M_2 then both processes deliver the two messages in the same order.

The identifier of an atomic broadcast message includes the protocol type (A_MSG), the message number (num) and the sender identifier (i).

The atomic broadcast protocol is implemented on top of the vector consensus protocol. It could also be implemented directly on top of the multi-valued consensus but, in the end, the functionality of the vector consensus protocol would have to be implemented in the protocol anyway. The approach we use is more modular and elegant, besides providing the two protocols, either of which may be useful for the system designer.

5.1. The protocol

The protocol is presented in Algorithm 3. It is inspired from the algorithms of Chandra, Hadzilacos and Toueg [18, 30], which assume crash faults. The initialization is carried out before the first transmission or reception of a message (lines 1–4). A process atomically broadcasts a message by calling the procedure `A_Broadcast`, which simply reliably broadcasts the message to all processes (lines 5 and 6). The message number num guarantees that all messages broadcast by a correct process are unique, since this number is unique. If a malicious process tries to call `R_Broadcast` twice with the same message, then the reliable broadcast protocol delivers the message only once (see property RB3, Integrity).

The delivery of messages is handled by tasks T1 and T2. When a message is delivered by the reliable broadcast protocol, it is inserted in the set R_delivered$_i$ (lines 15 and 16). Whenever this set is not empty, the process tries to agree with the other processes on the delivery of the messages in the set (lines 7–14). The task starts by constructing a vector H$_i$ with a *hash* of each of the messages in R_delivered$_i$ (line 8). A hash works essentially as a fixed-length unique identifier of the message. The objective is to compress the input supplied to the vector consensus protocol, since the performance of this protocol depends on the size of the value (e.g. the communication time depends on the size of the messages). A hash is obtained using a *hash function h* defined by the following properties [33]:

- *HF1 Compression: h* maps an input x of arbitrary finite length, to an output $h(x)$ of fixed length.
- *HF2 One way:* for all pre-specified outputs, it is computationally infeasible to find an input that hashes to that output.
- *HF3 Weak collision resistance:* it is computationally infeasible to find any second input that has the same output as a specified input.[5]
- *HF4 Strong collision resistance:* it is computationally infeasible to find two different inputs that hash to the same output.

The value proposed by a process to the vector consensus is itself a vector with the hashes of the messages, H$_i$ (lines 8 and 9). The vector consensus protocol decides on a vector X$_i$ with at least $(2f + 1)$ vectors H from different processes. If the hash of a message appears in at least $(f + 1)$ of these vectors, the process can be confident that the hash was proposed by at least one correct process (there are at most f corrupt processes); therefore there is no doubt that the message was reliably broadcast to all processes. This is important because a malicious process might provide a hash for which there was no message to deliver. The process waits until all messages that are to be delivered are put in R_delivered$_i$ (line 10), then it stores them in A_deliver$_i$ (line 10). Finally, the process delivers the messages in A_deliver$_i$ in a pre-established order, removes them from R_delivered$_i$, and increments the atomic broadcast identifier (lines 12–14).

[5]A guessing attack is expected to break the property HF3 in 2^m hashing operations, where m is the number of bits of the hash. A birthday attack can be expected to break property HF4 in $2^{m/2}$ hashing operations. In a practical setting, a hashing function with 160 bits like SHA-1 [37] can be considered secure enough for our protocol. Nevertheless, we consider HF2, HF3 and HF4 to be assumptions.

ALGORITHM 3. Atomic broadcast protocol (for process p_i).

```
Initialization:
 1: R_delivered_i ← ∅;   {messages delivered by the reliable broadcast protocol}
 2: aid_i ← 0;            {atomic broadcast identifier}
 3: num_i ← 0;            {message number}
 4: activate task (T1,T2);

When Procedure A_Broadcast (m) is called do
 5: R_Broadcast ( ⟨A_MSG, num_i, m, i⟩ );
 6: num_i ← num_i + 1;

Task T1:
 7: when (R_delivered_i ≠ ∅) do
 8:     H_i ← {hashes of the messages in R_delivered_i};
 9:     X_i ← Vector_Consensus (H_i, aid_i);
10:     wait until (all messages with hash in f + 1 or more cells in vector X_i are in R_delivered_i);
11:     A_deliver_i ← {all messages with hash in f + 1 or more cells in vector X_i};
12:     atomically deliver messages in A_deliver_i in a deterministic order;
13:     R_delivered_i ← R_delivered_i - A_deliver_i;
14:     aid_i ← aid_i + 1;

Task T2:
15: when ⟨A_MSG, num, m, i⟩ is delivered by the reliable broadcast protocol do
16:     R_delivered_i ← R_delivered_i ⋃ {⟨A_MSG, num, m, i⟩};
```

5.2. Correctness proof

The atomic broadcast protocol in Algorithm 3 is correct if it satisfies the properties AB1, AB2, AB3 and AB4.

THEOREM 11. (Validity). *If a correct process broadcasts a message M, then some correct process eventually delivers M.*

Proof. A correct process broadcasts a message M by calling `A_Broadcast(m)`. Then, the atomic broadcast protocol adds a header to the message and broadcasts it using the reliable broadcast protocol (line 5). The properties of this reliable broadcast protocol ensure that all correct processes eventually receive M (properties RB1–RB3). This guarantees that there is an execution of the lines 7–14 when all correct processes put the hash of M in H (line 8), unless these processes already delivered M in a previous execution of line 12. When all correct processes put the hash of M in H, the vector consensus decides on a vector that includes at least $f + 1$ entries with that hash (property VC1, Vector validity). Therefore, if the protocol does not block, all correct processes deliver M (lines 10–12).

The protocol might block only in lines 9 and 10. It does not block in line 9 because the vector consensus is guaranteed to terminate (property VC3, Termination). Line 10 waits until all messages that have to be delivered by the atomic broadcast protocol (those with $f + 1$ hashes in the vector) are in R_delivered. A message with $f + 1$ hashes in the vector must have been already delivered by the reliable broadcast protocol to at least one correct process. Therefore, this protocol will eventually deliver the message to all correct processes (properties RB1–RB3), so no correct process blocks in line 10. □

THEOREM 12. (Agreement). *If a correct process delivers a message M, then all correct processes eventually deliver M.*

Proof. The theorem assumes that one correct process, say p_i, delivers M. Therefore: (i) the vector consensus in line 9 decides on a vector with at least $f + 1$ hashes of M; and (ii) the reliable broadcast protocol delivers M to p_i; therefore it delivers M to all correct processes (properties RB1–RB3). All correct processes get the same results from the vector consensus so all eventually deliver M. □

THEOREM 13. (Integrity). *For any message M, every correct process p delivers M at most once, and if sender(M) is correct then M was previously broadcast by sender(M).*

Proof. The proof of the first assertion is trivial from the inspection of the algorithm, assuming the properties of hash functions. The proof of the second assertion follows directly from the properties of the communication channels. □

THEOREM 14. (Total order). *If two correct processes deliver two messages M_1 and M_2 then both processes deliver the two messages in the same order.*

Proof. Any correct process delivers messages only after an execution of `Vector_Consensus` (line 9). All correct processes execute the same instances of the vector consensus protocol, identified by aid = 0, 1, 2, ... The messages which are delivered are all those with at least $f + 1$ hashes in the vector returned by `Vector_Consensus` and the order of delivery is deterministic (line 12). Therefore, all processes deliver the same messages in the same order. □

6. MULTI-VALUED CONSENSUS AND ATOMIC BROADCAST EQUIVALENCE

The equivalence between crash-tolerant multi-valued consensus and atomic broadcast has been proved in [18, 30]. The equivalence for environments prone to Byzantine faults with signatures has been proved in [31]. Here we prove a similar result but without the requirement of signatures. This result has been previously stated but never proved [18, 30].

We follow an approach similar to [18, 30], i.e. we provide a transformation from multi-valued consensus (as defined in Section 3) to atomic broadcast and a transformation from atomic broadcast to multi-valued consensus. The first transformation was, in fact, presented in two steps in Sections 4 and 5. The transformation from atomic broadcast to consensus is presented in Algorithm 4. The transformations are independent of the technique used to circumvent FLP.

ALGORITHM 4. Transformation from atomic broadcast to multi-valued consensus (for process p_i).

```
Function M_V_Consensus_AB ($v_i$, cid)
1: INIT_delivered$_i$ $\leftarrow \emptyset$;          {INIT messages delivered}
2: A_Broadcast ( $\langle$INIT, $v_i$, cid, i$\rangle$ ); {atomic broadcast}
3: wait until (at least $(n-f)$ INIT messages from different
   senders have been atomically delivered);
4: $\forall_j$: if ($\langle$INIT, $v_j$, cid, j$\rangle$ has been delivered) then $V_i$[j]
   $\leftarrow v_j$; else $V_i$[j] $\leftarrow \perp$;
5: if ($\exists_v : \#_v(V_i) \geq (n-2f)$) then
6:    return v;
7: else
8:    return $\perp$;
```

The protocol is similar to the first part of Algorithm 1 so there is no need to describe its behavior. The protocol is correct if it satisfies the properties MVC1 through MVC5 provided in Section 3.

THEOREM 15. (Validity 1). *If all correct processes propose the same value v, then any correct process that decides, decides v.*

Proof. If all correct processes propose the same value v, then all processes deliver at least $(n - 2f)$ INIT messages with v in line 3 since at most f processes can broadcast messages with different values. It follows immediately from lines 5 and 6 that any correct process that decides, decides v. □

THEOREM 16. (Validity 2). *If a correct process decides v, then v was proposed by some process or $v = \perp$.*

Proof. The proof is obtained from a trivial inspection of the protocol. □

THEOREM 17. (Validity 3). *If a value v is proposed only by corrupt processes, then no correct process that decides, decides v.*

Proof. For a correct process to decide v (line 6), at least $(n - 2f)$ processes must have broadcast that value. There can be at most $f < (n - 2f)$ corrupt processes so no correct processes can decide a value proposed only by those processes. □

THEOREM 18. (Agreement). *No two correct processes decide differently.*

Proof. The atomic broadcast protocol guarantees that all correct processes deliver the INIT messages in the same order. Therefore, all correct processes deliver the same INIT messages in line 3 and decide the same in lines 5–8. □

THEOREM 19. (Termination). *Every correct process eventually decides.*

ALGORITHM 5. Transformation from vector consensus to multi-valued consensus (for process p_i).

```
Function M_V_Consensus_VC (v_i, cid)
1: $V_i \leftarrow$ Vector_Consensus (v_i, cid);
2: if ($\exists_v : \#_v(V_i) \geq (n - 2f)$) then
3:     return v;
4: else
5:     return $\perp$;
```

Proof. The proof is trivial taking into account that the atomic broadcast protocol terminates (properties AB1 and AB2) and that there are at least $(n - f)$ correct processes. □

The proof that Algorithm 4 satisfies the definition of multi-valued consensus concludes the demonstration that atomic broadcast and multi-valued consensus are equivalent. An immediate consequence is that the FLP impossibility result also applies to Byzantine-resilient atomic broadcast, i.e. this problem cannot be solved deterministically in asynchronous systems. The protocol shown in this paper circumvents this result using randomization, i.e. by not being deterministic.

7. MULTI-VALUED CONSENSUS AND VECTOR CONSENSUS EQUIVALENCE

Vector consensus is apparently a stronger problem than consensus. Doudou and Schiper proved that a flavor of multi-valued consensus defined in terms of properties MVC1/MVC4/MVC5 is reducible to vector consensus [20]. Here we prove that a multi-valued consensus defined by properties MVC1–MVC5 is equivalent to vector consensus. The transformation from multi-valued consensus to vector consensus was given in Section 4. The reverse transformation is shown in Algorithm 5. We skip the correctness proof of this transformation given its simplicity. The two transformations together prove the equivalence of the two problems.

8. PERFORMANCE EVALUATION

Multi-valued consensus. The time complexity of the multi-valued consensus protocol is twice the number of *asynchronous rounds* executed by the reliable broadcast protocol L_{rb} (lines 3 and 10) plus the time complexity of the binary consensus protocol L_{bc} (line 17). The reliable broadcast protocol by Bracha runs in exactly three rounds [9]. The time complexity of the binary consensus protocol is measured in *expected number of asynchronous rounds*, since the protocol is randomized, therefore probabilistic. The binary consensus protocol in Appendix A has constant expected time complexity $O(1)$, or, more precisely, $L_{\text{bc}} = 20$. The protocol by Canetti and Rabin has also constant expected time but has a high message complexity so we do not consider it here [10].[6] Therefore, the time complexity of the multi-valued consensus protocol is (we use capital L for expected number of asynchronous rounds):

$$L_{\text{mvc}} = 2L_{\text{rb}} + L_{\text{bc}} = 26 = O(1) \tag{1}$$

The protocol can be optimized by replacing the second reliable broadcast in line 10 by a (normal) broadcast or by the transmission of the VECT message individually to all processes. In this case, one correct process might receive $(n - 2f)$ messages with the value to be decided v, while another correct process would not. To circumvent this problem, all correct processes that receive $(n-2f)$ messages with the value v (line 11) have to resend these messages to all other processes. This optimization reduces the three rounds of the reliable broadcast protocol to two rounds.

Table 1 presents both the expected time complexity of the protocol (L_{mvc}) and the time complexity in the best case (l_{mvc}). The best case for the multi-valued consensus protocol is when the binary consensus runs in $l_{\text{bc}} = 10$ rounds instead of the expected $L_{\text{bc}} = 20$ rounds (see Appendix A). Notice that the reliable broadcast runs in a constant number of rounds, therefore $l_{\text{rb}} = L_{\text{rb}} = 3$.

Message complexities differ if the communication is *point-to-point* or *broadcast*. If the communication is point-to-point, the message complexity of Bracha's reliable broadcast is $M_{\text{rb}} = 2n^2 + n$ and the expected message complexity of the binary consensus in the appendix is $M_{\text{bc}} = 12n^3 + 8n^2$. If the messages are broadcast, these complexities are respectively: $M'_{\text{rb}} = 2n + 1$ and $M'_{\text{bc}} = 12n^2 + 8n$. The expected message complexity of our multi-valued consensus corresponds to $2n$ executions of the reliable broadcast plus one binary consensus (Table 2):

$$M_{\text{mvc}} = 2nM_{\text{rb}} + M_{\text{bc}} = 16n^3 + 10n^2 = O(n^3) \tag{2}$$

$$M'_{\text{mvc}} = 2nM'_{\text{rb}} + M'_{\text{bc}} = 16n^2 + 10n = O(n^2) \tag{3}$$

These complexities can be reduced by merging or piggy-backing some messages in others.

Vector consensus. The vector consensus protocol runs in the best case in one round, in the worst in $f + 1$ rounds (e.g. if $n = 4, f = 1$, the protocol terminates in one or two rounds). In the best case the loop in lines 3–8 will be executed only once so the time complexity will be the sum of those of the reliable broadcast (line 2) and the multi-valued consensus (line 6). If the protocol does not terminate in the end of the first round, it is reasonable to expect that all VC_INIT messages reliably broadcast will be delivered during the first execution of `M_V_Consensus`, since this consensus involves several rounds of message exchange (two reliable broadcasts plus one binary consensus). This would make the protocol terminate in the second round. However, if we make the (pessimistic) assumption that the malicious processes control the communication, then they can schedule the messages in such a way that they delay the protocol a maximum of f rounds. Therefore, the expected time complexity of the algorithm is $O(f)$:

$$L_{\text{vc}} = L_{\text{rb}} + (f + 1)L_{\text{mvc}} = O(f) \tag{4}$$

[6]The binary consensus protocol by Bracha has also an expected number of rounds of $O(1)$ if $f = O(\sqrt{n})$, but $O(2^{n-f})$ otherwise [9].

TABLE 1. Time complexities of the three protocols (asynchronous rounds).

Protocol	Best time complexity	Expected time complexity
Multi-valued consensus	$l_{\mathrm{mvc}} = 2l_{\mathrm{rb}} + l_{\mathrm{bc}} = 16$	$L_{\mathrm{mvc}} = 2L_{\mathrm{rb}} + L_{\mathrm{bc}} = 26 = O(1)$
Vector consensus	$l_{\mathrm{vc}} = l_{\mathrm{rb}} + l_{\mathrm{mvc}} = 19$	$L_{\mathrm{vc}} = L_{\mathrm{rb}} + (f+1)L_{\mathrm{mvc}} = O(f)$
Atomic broadcast	$l_{\mathrm{ab}} = l_{\mathrm{rb}} + l_{\mathrm{vc}} = 22$	$L_{\mathrm{ab}} = L_{\mathrm{rb}} + L_{\mathrm{vc}} = O(f)$

TABLE 2. Message complexities of the three protocols (messages).

Protocol	Expected message complexity (point-to-point)	Expected message complexity (broadcast)
Multi-valued consensus	$M_{\mathrm{mvc}} = 2nM_{\mathrm{rb}} + M_{\mathrm{bc}} = O(n^3)$	$M'_{\mathrm{mvc}} = 2nM'_{\mathrm{rb}} + M'_{\mathrm{bc}} = O(n^2)$
Vector consensus	$M_{\mathrm{vc}} = nM_{\mathrm{rb}} + (f+1)M_{\mathrm{mvc}} = O(fn^3)$	$M'_{\mathrm{vc}} = nM'_{\mathrm{rb}} + (f+1)M'_{\mathrm{mvc}} = O(fn^2)$
Atomic broadcast	$M_{\mathrm{ab}} = M_{\mathrm{rb}} + M_{\mathrm{vc}} = O(fn^3)$	$M'_{\mathrm{ab}} = M'_{\mathrm{rb}} + M'_{\mathrm{vc}} = O(fn^2)$

The best case is the execution of a single multi-valued consensus with an execution of the best case of the binary consensus:

$$l_{\mathrm{vc}} = l_{\mathrm{rb}} + l_{\mathrm{mvc}} = 19 \tag{5}$$

The expected message complexities correspond to n executions of the reliable broadcast plus $f+1$ multi-valued consensuses:

$$M_{\mathrm{vc}} = nM_{\mathrm{rb}} + (f+1)M_{\mathrm{mvc}} = 18n^3 + 11n^2 + 16n^3 f + 10n^2 f = O(fn^3) \tag{6}$$

$$M'_{\mathrm{vc}} = nM'_{\mathrm{rb}} + (f+1)M'_{\mathrm{mvc}} = 18n^2 + 11n + 16n^2 f + 10nf = O(fn^2) \tag{7}$$

Atomic broadcast. The time complexity of the atomic broadcast protocol is equivalent to one reliable broadcast (line 5) plus one vector consensus (line 9); therefore the expected number of rounds is $O(f)$ per message:

$$L_{\mathrm{ab}} = L_{\mathrm{rb}} + L_{\mathrm{vc}} = O(f) \tag{8}$$

The best time complexity is one reliable broadcast plus a best case execution of the vector consensus:

$$l_{\mathrm{ab}} = l_{\mathrm{rb}} + l_{\mathrm{vc}} = 22 \tag{9}$$

The expected message complexities depends on the amount of messages being transmitted. If only occasional messages are sent, the expected message complexities are respectively with point-to-point and broadcast communication:

$$M_{\mathrm{ab}} = M_{\mathrm{rb}} + M_{\mathrm{vc}} = 18n^3 + 13n^2 + n + 16n^3 f + 10n^2 f = O(fn^3) \tag{10}$$

$$M'_{\mathrm{ab}} = M'_{\mathrm{rb}} + M'_{\mathrm{vc}} = 18n^2 + 13n + 1 + 16n^2 f + 10nf = O(fn^2) \tag{11}$$

However, if messages go on arriving during a certain execution of the vector consensus protocol, in the next round task T1 will try to make agreement on several messages instead of only one. Therefore this protocol exhibits the virtuous characteristic that its number of messages decline considerably if the rate of transmissions increases.

Tables 1 and 2 summarize the results for all protocols.

9. RELATED WORK

The FLP impossibility result implies that any consensus protocol in a strictly asynchronous environment has to be randomized. Most randomized consensus protocols presented in the literature are binary. An exception is the multi-valued crash-tolerant protocol in [38]. Also for crash failures, there is one transformation from binary to multi-valued consensus available [39]. Turpin and Coan presented a transformation from binary to multi-valued consensus for Byzantine synchronous systems [40]. Toueg presented a transformation for asynchronous systems [12]. The main difference of this transformation to Algorithm 1 is that Toueg uses signatures; therefore its algorithm does not require a reliable broadcast primitive but a weaker echo broadcast protocol. His protocol has optimal resilience, time complexity $O(1)$, and lower message complexity than ours, but needs asymmetric cryptography. Cachin *et al.* [31] proposed a similar transformation, but the algorithm is based on voting the selection of the value proposed by each successive process. The protocol has optimal resilience, time complexity $O(1)$ and lower message complexity but uses public-key signatures and threshold cryptography. Several non-randomized, Byzantine-resilient, asynchronous multi-valued consensus protocols have been proposed in the literature [15, 21, 22, 34, 35]. Lower bounds on the number of rounds necessary for (Byzantine) consensus and atomic broadcast have been defined in [41].

Interactive consistency was defined as the problem of agreeing on a vector with one value per correct process [1]. However, in asynchronous systems it is not possible to differentiate slow from crashed processes, and with a Byzantine fault model it might also be impossible to distinguish malicious from crashed processes. Therefore, for Byzantine asynchronous systems the vector consensus problem was defined [20]. Two vector consensus protocols based on failure detectors and one based in wormholes have been specified [20, 21, 26]. Recently, Ben-Or and El-Yaniv presented a randomized vector consensus protocol with optimal resilience, time complexity $O(1)$ and no signatures [32]. However, the message complexity is considerably higher than ours, since the protocol runs n multi-valued consensus protocols in parallel, while ours runs, in the worst case, $n - (2f + 1) + 1$ multi-valued consensuses.

For the crash fault model, some transformations from multi-valued consensus to atomic broadcast have been defined [18, 30, 36]. Cachin *et al.* [31] defined a transformation from multi-valued consensus to atomic broadcast for Byzantine faults with signatures. Doudou *et al.* [22] presented a transformation closer to ours. It also uses signatures and it can have a higher communication complexity since it gives the full messages to the consensus module, instead of hashes, which are generally smaller. Doudou and Schiper briefly discuss a reduction of atomic broadcast to vector consensus [20].

A collection of randomized atomic broadcast protocols can be found in [42]. These protocols rely on signatures to guarantee the authenticity of the messages and do not have optimal resilience. Other Byzantine-resistant atomic broadcasts for asynchronous systems can be found in Rampart [19] that uses signatures and SecureRing [43] that uses a signed token. BFT [6] does not use signatures when there are no faults; therefore it is very efficient. Unlike ours, all these three protocols need a failure detector to put away corrupt processes. Apart from the added complexity, the design of Byzantine failure detectors that are complete is still an open research issue. Défago *et al.* [44] present an interesting classification of atomic broadcast protocols. In terms of that classification, our protocol is a *destination agreement* algorithm, i.e. processes receive messages without ordering information and run agreements to order them.

10. CONCLUSION

This paper proposes a stack of intrusion-tolerant or Byzantine-resistant protocols. These protocols form a coherent family, sharing effective and efficient structural properties: signature freedom, full asynchrony, decentralization and optimal resilience.

The stack shows a series of protocol transformations: from binary consensus to multi-valued consensus, from multi-valued consensus to vector consensus and from vector consensus to atomic broadcast. The objective is to provide a modular set of protocols that a designer can use in practice in the construction of intrusion-tolerant systems, especially in systems with limited resources like embedded environments. Therefore, the protocols evade a set of characteristics that might constitute a shortcoming in a real system: the use of public-key signatures, a known performance bottleneck in intrusion-tolerant systems, time assumptions, often vulnerable to some attacks and the existence of leaders whose failure might be costly to detect.

The multi-valued consensus protocol terminates in a constant expected number of rounds. However, due to the severe nature of malicious faults, vector consensus is more effective as a system building block for security-related applications. The time complexity of the vector consensus proposed is $O(f)$. The time complexity of the atomic broadcast protocol is also $O(f)$ (per message), although the average number of rounds can be considerably lower if there are several messages being transmitted. Both the time complexities of the vector consensus and atomic broadcast protocols are reduced to $O(1)$ when all processes are correct. These results look very promising.

Besides presenting the stack of protocols, the paper also proves the equivalence between multi-valued consensus and atomic broadcast in the Byzantine failure model without signatures. A similar proof is given for the equivalence between multi-valued consensus and vector consensus.

ACKNOWLEDGEMENTS

We warmly thank the anonymous reviewers for their comments that greatly assisted us in improving the paper. This work was partially supported by the EU through project CRitical UTility InfrastructurAL Resilience (CRUTIAL), and the FCT through project POSI/EIA/60334/2004 (RITAS) and the Large-Scale Informatic Systems Laboratory (LASIGE).

REFERENCES

[1] Pease, M., Shostak, R. and Lamport, L. (1980) Reaching agreement in the presence of faults. *J. ACM*, **27**, 228–234.

[2] Lamport, L., Shostak, R. and Pease, M. (1982) The Byzantine generals problem. *ACM Trans. Progr. Lang. Syst.*, **4**, 382–401.

[3] Rabin, M. O. (1983) Randomized Byzantine generals. In *Proc. 24th Annual IEEE Symp. Foundations of Computer Science*, Tucson, AZ, November 7–9, pp. 403–409. IEEE Computer Society Press, Los Alamitos, USA.

[4] Ben-Or, M. (1983) Another advantage of free choice: completely asynchronous agreement protocols. In *Proc. 2nd ACM Symp. Principles of Distributed Computing*, Montreal, Canada, August 17–19, pp. 27–30. ACM Press, New York, USA.

[5] Veríssimo, P., Neves, N. F. and Correia, M. (2003) Intrusion-tolerant architectures: concepts and design. In Lemos, R., Gacek, C. and Romanovsky, A. (eds), *Architecting Dependable Systems*, *LNCS*, **2677**, pp. 3–36. Springer-Verlag, London, UK.

[6] Castro, M. and Liskov, B. (2002) Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, **20**, 398–461.

[7] Schneider, F. B. (1990) Implementing faul-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, **22**, 299–319.

[8] Reiter, M. K. (1995) The Rampart toolkit for building high-integrity services. In *Theory and Practice in Distributed Systems*, *LNCS*, **938**, pp. 99–110. Springer-Verlag, London, UK.

[9] Bracha, G. (1984) An asynchronous $\lfloor (n-1)/3 \rfloor$-resilient consensus protocol. In *Proc. 3rd ACM Symp. Principles of Distributed Computing*, Vancouver, Canada, August 27–29, pp. 154–162. ACM Press, New York, USA.

[10] Canetti, R. and Rabin, T. (1993) Fast asynchronous Byzantine agreement with optimal resilience. In *Proc. 25th Annual ACM Symp. Theory of Computing*, San Diego, USA, May 16–18, pp. 42–51. ACM Press, New York, USA.

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

[12] Toueg, S. (1984) Randomized Byzantine agreements. In *Proc. 3rd ACM Symp. Principles of Distributed Computing*, Vancouver, Canada, August 27–29, pp. 163–178. ACM Press, New York, USA.

[13] Bracha, G. and Toueg, S. (1985) Asynchronous consensus and broadcast protocols. *J. ACM*, **32**, 824–840.

[14] Cachin, C., Kursawe, K. and Shoup, V. (2000) Random oracles in Constantinople: practical asynchronous Byzantine agreement using cryptography. In *Proc. 19th ACM Symp. Principles of Distributed Computing*, Portland, USA, July 16–19, pp. 123–132. ACM Press, New York, USA.

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

[16] Fetzer, C. and Cristian, F. (1995) On the possibility of consensus in asynchronous systems. In *Proc. Pacific Rim Int. Symp. Fault-Tolerant Systems*, Newport Beach, USA, December 4–5, pp. 86–91. IEEE Computer Society Press, Los Alamitos, CA.

[17] Veríssimo, P. and Almeida, C. (1995) Quasi-synchronism: a step away from the traditional fault-tolerant real-time system models. *Bullettin of the Technical Committee on Operating Systems and Application Environments*, **7**, 35–39.

[18] Chandra, T. and Toueg, S. (1996) Unreliable failure detectors for reliable distributed systems. *J. ACM*, **43**, 225–267.

[19] Reiter, M. (1994) Secure agreement protocols: reliable and atomic group multicast in Rampart. In *Proc. 2nd ACM Conf. Computer and Communications Security*, Fairfax, USA, November 2–4, pp. 68–80. ACM Press, New York, USA.

[20] Doudou, A. and Schiper, A. (1997) *Muteness Detectors for Consensus with Byzantine Processes*. TR DSC 1997-30. École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland.

[21] Baldoni, R., Helary, J., Raynal, M. and Tanguy, L. (2000) Consensus in Byzantine asynchronous systems. In *Proc. Int. Colloquium on Structural Information and Communication Complexity*, L'Aquila, Italy, June 20–22, pp. 1–16. Carleton Scientific, Waterloo, Canada.

[22] Doudou, A., Garbinato, B. and Guerraoui, R. (2002) Encapsulating failure detection: from crash-stop to Byzantine failures. In Blieberger, J. and Strohmeier, A. (eds), *Int. Conf. Reliable Software Technologies, Ada-Europe*, June 17–21, *LNCS*, **2361**, pp. 24–50. Springer-Verlag, Berlin, Germany.

[23] Pedone, F., Schiper, A., Urbán, P. and Cavin, D. (2002) Solving agreement problems with weak ordering oracles. In Grandoni, F. and Thévenod-Fosse, P. (eds), *Proc. Fourth European Dependable Computing Conf.*, Parc des Expositions, Toulouse, France, October 23–25, *LNCS*, **2485**, pp. 44–61. Springer-Verlag, London, UK.

[24] Correia, M., Neves, N. F., Lung, L. C. and Veríssimo, P. (2005) Low complexity Byzantine-resilient consensus. *Distrib. Comput.*, **17**, 237–249.

[25] Neves, N. F., Correia, M. and Veríssimo, P. (2004) Wormhole-aware Byzantine protocols. In *2nd Bertinoro Workshop on Future Directions in Distributed Computing: Survivability, Obstacles and Solutions*, Bertinoro, Italy, June 23–25.

[26] Neves, N. F., Correia, M. and Veríssimo, P. (2005) Solving vector consensus with a wormhole. *IEEE Trans. Parall. Distrib. Sys.*, **16**, 1120–1130.

[27] Mostefaoui, A., Rajsbaum, S. and Raynal, M. (2003) Conditions on input vectors for consensus solvability in asynchronous distributed systems. *J. ACM*, **50**, 922–954.

[28] Friedman, R., Mostefaoui, A., Rajsbaum, S. and Raynal, M. (2002) Distributed agreement and its relation with error-correcting codes. In Malkhi, D. (ed.), *Proc. 16th Int. Conf. Distributed Computing*, Toulouse, France, October 28–30, *LNCS*, **2508**, pp. 63–87. Springer-Verlag, London, UK.

[29] Guerraoui, R. and Schiper, A. (1997) Consensus: the big misunderstanding. In *Proc. IEEE Int. Workshop on Future Trends in Distributed Computing Systems*, Tunis, Tunisia, October, pp. 183–188. IEEE Computer Society Press, Washington, USA.

[30] Hadzilacos, V. and Toueg, S. (1994) *A Modular Approach to Fault-Tolerant Broadcasts and Related Problems*. Technical Report TR94-1425, Department of Computer Science, Cornell University, Ithaca, USA.

[31] Cachin, C., Kursawe, K., Petzold, F. and Shoup, V. (2001) Secure and efficient asynchronous broadcast protocols (extended abstract). In Kilian, J. (ed.), *Advances in Cryptology: CRYPTO 2001*, Santa Barbara, CA, August 19–23, *LNCS*, **2139**, pp. 524–541. Springer-Verlag, New York, USA.

[32] Ben-Or, M. and El-Yaniv, R. (2003) Optimally-resilient interactive consistency in constant time. *Distrib. Comput.*, **16**, 249–262.

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

[34] Malkhi, D. and Reiter, M. (1997) Unreliable intrusion detection in distributed computations. In *Proc. 10th Computer Security Foundations Workshop*, Rockport, USA, June 10–12, pp. 116–124. IEEE Computer Society Press, Los Alamitos, USA.

[35] Kihlstrom, K. P., Moser, L. E. and Melliar-Smith, P. M. (2003) Byzantine fault detectors for solving consensus. *Comput. J.*, **46**, 16–35.

[36] Guerraoui, R. and Schiper, A. (2001) The generic consensus service. *IEEE Trans. Softw. Eng.*, **27**, 29–41.

[37] NIST (1994) *Announcement of weakness in the secure hash standard*. National Institute of Standards and Technology, Gaithersburg, USA.

[38] Ezhilchelvan, P., Mostefaoui, A. and Raynal, M. (2001) Randomized multivalued consensus. In *Proc. 4th IEEE Int. Symp. Object-Oriented Real-Time Computing*, Magdeburg, Germany, May 2–4, pp. 195–200. IEEE Computer Society Press, Los Alamitos, USA.

[39] Mostefaoui, A., Raynal, M. and Tronel, F. (2000) From binary consensus to multivalued consensus in asynchronous message-passing systems. *Inform. Process. Lett.*, **73**, 207–212.

[40] Turpin, R. and Coan, B. A. (1984) Extending binary Byzantine agreement to multivalued Byzantine agreement. *Inform. Process. Lett.*, **18**, 73–76.

[41] Dutta, P., Guerraoui, R. and Vukolic, M. (2005) *Best-Case Complexity of Asynchronous Byzantine Consensus*. Technical Report 200499, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland.

[42] Moser, L. E. and Melliar-Smith, P. M. (1999) Byzantine-resistant total ordering algorithms. *Inform. Comput.*, **150**, 75–111.

[43] Kihlstrom, K. P., Moser, L. E. and Melliar-Smith, P. M. (2001) The SecureRing group communication system. *ACM Trans. Inform. Syst. Security*, **4**, 371–406.

[44] Défago, X., Schiper, A. and Urbán, P. (2004) Totally ordered broadcast and multicast algorithms: taxonomy and survey. *ACM Comput. Surv.*, **36**, 372–421.

A. A BINARY CONSENSUS PROTOCOL

This appendix presents a Binary consensus protocol compatible with the properties we stated in Section 1: it does not use signatures; it is asynchronous (uses randomization to circumvent FLP); decisions are taken in a decentralized way during the normal operation; it has optimal resilience, $f = \lfloor (n-1)/3 \rfloor$. Moreover, its time complexity is $O(1)$. The protocol is a version of Bracha's protocol in [9] enhanced with the *dual-threshold coin-tossing scheme* by Cachin *et al.* [14]. The protocol does not avoid public-key cryptography entirely since the coin-tossing scheme is based on the Diffie–Hellman problem.

The *(n,k,f) dual-threshold coin-tossing scheme* assumes n processes, at most f of which can be corrupt. The processes hold *shares of a function F* mapping a coin name C to its value $F(C) \in \{0, 1\}$. The main property of the scheme is that to construct the value of a coin, a process needs *k coin shares* from different processes, with $t < k \leq n - f$. Here we consider the specific case of $k = n - f$.

The scheme assumes a *trusted dealer* that generates secret keys $\mathrm{SK}_1, \ldots, \mathrm{SK}_n$ and verification keys VK, $\mathrm{VK}_1, \ldots, \mathrm{VK}_n$. The dealer gives every process p_i a secret key SK_i and all verification keys. A process uses SK_i to produce *coin shares* and the verification keys to construct the values of coins. The existence of the dealer does not collide with the protocol being decentralized (in the sense above), because the dealer has no role during the execution of the protocol.

The modification of Bracha's protocol is simple. Lets us define a coin name C as a unique combination of the consensus execution identifier $bcid$ and the round number r, e.g. $C = bcid + 1/r$. In Step 3, the protocol may have to set a variable i_p to 1 or 0 with probability 1/2 [9]. The modification is to use the dual-threshold coin-tossing scheme to give identical random numbers to all correct processes, i.e. coins with name C. More precisely, the line of Bracha's protocol that sets i_p to 1 or 0 is substituted by Step 4 of the ABBA protocol [14]. After that step, i_p is set to the value of coin C.

This protocol avoids the use of digital signatures and threshold signatures of the original protocol in [14] at the cost of additional rounds of message exchange. However, the expected time complexity is still $O(1)$, or more precisely (considering the reliable broadcast in [9]):

$$L_{\mathrm{bc}} = 6L_{\mathrm{rb}} + 2 = 20 \tag{12}$$

In the best case the protocol runs in a single round:

$$l_{\mathrm{bc}} = 3l_{\mathrm{rb}} + 2 = 10 \tag{13}$$

The expected message complexities are $M_{\mathrm{bc}} = 12n^3 + 8n^2 = O(n^3)$ with point-to-point communication or $M'_{\mathrm{bc}} = 12n^2 + 8n = O(n^2)$ with broadcast communication. However, several messages of the executed reliable broadcast might be merged or piggy-backed, thus reducing these numbers.