Modular Abstractions for Devising Byzantine-Resilient State Machine Replication

Assia Doudou*   Benoît Garbinato†   Rachid Guerraoui*

* Swiss Federal Institute of Technology	† Linkvest, Avenue des Baumettes 19
CH-1015 Lausanne (Switzerland)	CH-1020 Renens (Switzerland)
{*assia.doudou,rachid.guerraoui*}*@epfl.ch*	*benoit.garbinato@linkvest.com*

Abstract

State machine replication is a common approach for making a distributed service highly available and resilient to failures, by replicating it on different processes. It is well-known, however, that the difficulty of ensuring the safety and liveness of a replicated service increases significantly when no synchrony assumptions are made, and when processes can exhibit Byzantine behaviors. The contribution of this work is to break the complexity of devising a *Byzantine-resilient* state machine replication protocol, by decomposing it into key modular abstractions. In addition to being modular, the protocol we propose always preserves safety in presence of less than one third of Byzantine processes, independently of any synchrony assumptions. As for the liveness of our protocol, it relies on a Byzantine failure detector that encapsulates the sufficient amount of synchrony.

- Submission Categories: fault-tolerant systems, secure systems, distributed algorithms.

- Keywords: Byzantine Failures, Asynchronous Model, State Machine, Atomic Multicast, Weak Interactive Consistency, Failure Detector.

- Contact author: Assia Doudou

- Word Count: 9350

1 Introduction

A *state machine* is defined by a set of *state variables*, which encodes its state, and by a set of remotely accessible *commands*, which allow to transform its state. Each command is implemented by a deterministic program and is executed atomically with respect to other commands. A client issue a *request* to the state machine, which specifies the execution of a command; depending on the command, the state machine might return a reply to the client or not. Requests must be processed by the state machine sequentially, and in an order that is consistent with Lamport's causality relationship [10]. Intuitively, this relationship has two implications. First, the requests of each individual client must be processed in the order they were issued. Second, if there potentially exists a causal relationship between a request req made to the state machine by some client, and a subsequent request req' made by some other client, then req must be processed before req'.

1.1 Replicating A State Machine

Replication is a convenient approach for making a state machine highly available to its clients and for making it resilient to failures [10, 15]. It is well-known, however, that ensuring the consistency and the responsiveness of a replicated state machine is complex to achieve in presence of failures. This complexity further increases when no restrictions are made on the failure model, i.e., when assuming a Byzantine model, and when no assumptions are made on the synchrony of the system, i.e., when assuming an asynchronous model. This due to the FLP impossibility result[1] and to the fact that malicious processes can do whatever they want to prevent correct replicas from delivering their service.

Contribution. The contribution of this work is to break the complexity of devising a Byzantine-resilient active replication protocol, by decomposing it into key subproblems and by solving each one via a separate abstraction. This modularity allows us to identify additional synchrony assumptions under which the liveness of our protocol is ensured (FLP makes it impossible to ensure liveness without such additional assumptions). As for safety, our replication protocol always preserves it.[2] Finally, we sketch simple optimizations that make our protocol more efficient in failure-free runs, without compromising modularity.

Roughly speaking, our state machine replication protocol relies on an Atomic Multicast primitive which clients use to send their requests to server replicas. The Atomic Multicast protocol we propose is then decomposed into a Reliable Multicast protocol and a protocol that solves a new problem that we name *Weak Interactive Consistency*. This problem is a variation of the traditional Interactive Consistency problem [5]. Solving Weak Interactive Consistency in an asynchronous model also lead us to define a new class of failure detectors adapted to the Byzantine model: such failure detectors are at the heart of liveness issues. In this paper, however, we only sketch the problems of specifying and implementing failure detectors in a Byzantine model; a detailed discussion can be found in [3].

---

[1]FLP states that no algorithm can solve Consensus in an asynchronous system if one process can crash [4].

[2]By ensuring the safety and liveness of our active replication protocol, we guarantee the consistency and responsiveness of the replicated state machine.

1.2 Related Work

Rampart [12] and SecureRing protocols [7] are two research results on state machine replication in the Byzantine asynchronous model. A major difference with our work, however, lies in the fact that these systems heavily rely on membership protocols. Therefore, when some process is suspected (even falsely), a view change is triggered and the suspected process is excluded from the new view. By excluding a correct process, the guarantee that all correct processes remain in the same consistent state cannot be ensured. Indeed, correct processes that are falsely suspected are forced to become incorrect, and hence no longer need to satisfy safety. This results in protocols where even safety depends on synchrony assumptions, which is not the case of our state machine replication protocol.

A research result closer to ours is described in [1]. There, the authors present a state machine replication protocol based on the idea of sequenced views, but with *no exclusion of processes*. In each view, only one process (the primary) is responsible for ordering client requests. Similarly to ours, the safety of their protocol is ensured independently of synchrony assumptions. Contrary to our approach, however, liveness issues are not encapsulated in a well-defined abstraction, i.e., a failure detector. Instead, liveness relies on the use of timeouts to prevent correct processes from being blocked by a Byzantine primary, and on an assumption on the sequence of views. This assumption states that there will eventually exist a view with a correct primary that other correct processes will not time out. Furthermore, the replication protocol proposed in [1] is monolithic, i.e., with no structural decomposition into subproblems. This lack of intermediate finer-grain abstractions leads to a protocol that is difficult to understand or reason about.

1.3 Roadmap

The rest of the paper is organized as follows. Section 2 presents our system model, in particular it introduces the notion of so-called *muteness failures*, together with an associated failure detector. Then, Section 3 presents our Byzantine-resilient state machine replication protocol, based on an Atomic Multicast primitive. Section 4 describes how an Atomic Multicast primitive can be built by composing a Reliable Multicast abstraction and a Weak Interactive Consistency abstraction. Section 5 details our solution to the Weak Interactive Consistency problem, while Section 6 discusses safety and liveness issues of this solution. Section 7 closes the paper with remarks on Byzantine failures detectors and on possible optimizations of our replication protocol.

2   System Model

We now describe the system model that we assume throughout this work, and which is a key element in proving the correctness of our modular protocol suite. Note however that not all the assumptions presented below are necessary to prove the correctness of each and every protocol. The existence of an adequate failure detector, for example, is only assumed when devising and proving our Weak Interactive Consistency protocol.

Execution & Communication Model.   We consider a distributed system composed of a set of processes, N of which are servers (state machine replicas), and the remainder are clients. Processes communicate by message passing via a fully connected network composed

of reliable point-to-point channels, i.e., if p sends message m to q, and both q and p are correct, then q eventually receives m. In addition, these channels guarantee FIFO order and preserve the integrity of messages exchanged between correct processes; in particular, the content of messages cannot be altered by some active intruder. We assume no bounds on relative process speeds nor on communication delays, i.e., the system is considered to be *asynchronous*.

Byzantine Failure Model. We assume a *Byzantine failure model with message authentication* [9]. In such a model, processes can fail by crashing (i.e., prematurely stop participating in the protocol), but can also behave maliciously. An incorrect process can for instance send garbled and misleading messages, or can refuse to send expected messages. More generally, *Byzantine processes*, also known as *malicious processes*, can exhibit arbitrary behaviors. In contrast, a correct process executes an infinite number of steps, and respects the specification of the algorithm it is supposed to execute.

There exist however a limit to the power of malicious processes: thanks to message authentication, a malicious process cannot impersonate correct processes. Message authentication relies on the following *signature unforgeability assumption*: if a correct process p does not send a signed message m, then no correct process ever receives a message m correctly signed by p. In this paper, we assume that every correct process signs each of its messages before sending it.[3] In addition, we assume that the maximum number of Byzantine processes among the N servers is $f < N/3$. There is no restriction on the number of Byzantine clients.

Muteness Failure Detectors. As already suggested, our state machine replication protocol relies on a set of modular abstractions where distributed agreement plays a central role. So, in order to circumvent the FLP impossibility result, we augment our asynchronous distributed system with a so-called *muteness failure detector* $\Diamond M_{\mathcal{A}}$ [3]. This failure detector only captures a subset of all possible Byzantine behaviors, namely *muteness failures*.[4] Such failures are tightly bound to the algorithm $\mathcal{A}$ that is executed by correct processes, and they are at the heard of liveness issues.

Intuitively, muteness failures characterize faulty processes from which correct processes stop receiving $\mathcal{A}$ messages. For example, a muteness failure can occur when a Byzantine process simply crashes, or arbitrarily decides to stop communicating with some or all correct processes. More precisely, we say that a process q is mute, with respect to some algorithm $\mathcal{A}$ and some correct process p, if p stops receiving forever $\mathcal{A}$ messages from q. We can then formally specify our muteness failure detector $\Diamond M_{\mathcal{A}}$, by stating the $\mathcal{A}$-*completeness* and *Eventual weak $\mathcal{A}$-accuracy* properties that it satisfies. In this paper, $\Diamond M_{\mathcal{A}}$ is only used by our Weak Interactive Consistency protocol, i.e., this protocol plays the role of $\mathcal{A}$ in the properties given below.

Mute $\mathcal{A}$-completeness. There is a time after which every process that is mute to any correct process p, with respect to $\mathcal{A}$, is suspected to be mute by p forever.

Eventual weak $\mathcal{A}$-accuracy. There is a time after which some correct process p is no more suspected to be mute, with respect to $\mathcal{A}$, by any other correct process.

---

[3]Signature unforgeability can be implemented via public-key encryption techniques such as RSA [13].

[4]Yet, muteness failures are more general than crash failures.

3 Byzantine-Resilient State Machine Replication

When devising a state machine replication protocol, two ordering problems must be addressed: the preservation of causality between client requests, and the implementation of a total order on the processing of requests by the state machine replicas. Total order is usually ensured by having the clients use an Atomic Multicast primitive to send their requests to the replicated state machine; this is the approach that we adopt here. As for causal order, there are basically two approaches to preserve it. One consists in delegating this problem to the communication layer, i.e., to devise an Atomic Multicast that additionally ensures causal order. The other approach consists in delegating it to the application layer, i.e., to let the clients and the state machine replicas address this problem.

In this work, we adopt the second approach. More precisely, we force each client that issues a request to the replicated state machine to refrain from any other communication, as long as it does not receive a response to its current request; this solution was proposed by F.B. Schneider in [15]. Note that preserving causality between client requests at the application level has the advantage to avoid unnecessary ordering by the communication layer. Furthermore, devising a causal order multicast in the Byzantine failure model is still an open problem and it is not clear whether such a primitive would make sense or not [6].

In rest of this section, we start by formally specifying the Atomic Multicast problem. We then describe how a state machine replication protocol can be built on top of a primitive that solves this problem, in presence of malicious failures. A modular protocol to implement such an Atomic Multicast primitive in the Byzantine failure model is described in the next section.

3.1 Atomic Multicast

In this paper, we consider an Atomic Multicast primitive that allows clients to send messages to some static group g of N server replicas; this conforms to our system model assumptions. Furthermore, we assume that clients do not belong to g and that only replicas in g deliver the clients' messages. Each message m sent by a client contains the following fields: $seq(m)$, and $op(m)$. Field $seq(m)$ is a unique identifier associated with m and is composed of the client's identifier $seq.id(m)$ plus a sequence number, e.g., the local clock of the client. Consequently, the identity of a message is defined by the client that issues it. Finally, field $op(m)$ specifies the operation (including parameters) that the client wants to execute on the replicated state machine. Formally, Atomic Multicast is defined by two primitives: *A_multicast* and *A_deliver*, which allow to send and deliver messages according to the following properties.

- *Validity.* If a correct client c A_multicasts a message m to g, then some correct replica in g eventually A_delivers m.
- *Agreement.* If a message m is A_delivered by some correct replica in g, then all correct replicas in g eventually A_deliver m.
- *Integrity.* For any message m, every correct replica p A_delivers m at most once. Furthermore, if client q of m is correct, then no correct replica A_deliver m unless q has previously A_multicast m.
- *Total Order.* If correct replicas p and q both A_deliver messages m and m', then p A_delivers m before m' if and only if q A_delivers m before m'.

3.2   A Protocol for State Machine Replication

Based on the above Atomic Multicast communication primitive, our state machine replication protocol is rather simple. To execute some operation *op* on the replicated state machine, a correct client *c* A_multicasts a signed message *m* containing *op* to all processes in *g*. Then, *c* waits for $f+1$ correctly signed reply messages from distinct replicas in *g*, all with the same result. Waiting for $f+1$ similar replies ensures that at least one reply was sent by a correct replica and hence is valid. Until then, *c* does not issue any other communication.

On the server side, each correct state machine replica delivers the request message using the A_deliver primitive, and executes the operation *op* contained in this message. Then, a reply message containing the result of *op*'s execution is sent to the client from which the request message was issued, i.e., *seq.id(m)*.

4   A Modular Atomic Multicast Protocol

We now propose a modular Atomic Multicast protocol based on two Byzantine-resilient abstractions: Reliable Multicast and Weak Interactive Consistency (noted hereafter WIConsistency). The modularity of our approach is illustrated by the layered architecture depicted in Figure 1. The role of the Muteness Failure Detector, which is only used by the WIConsistency layer, will be explained in Section 5. In the following, we first present each abstraction independently and then we show how they cooperate to solve Atomic Multicast.
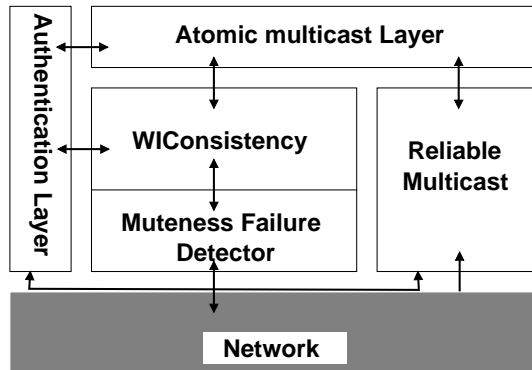

Figure 1: A Modular Composition of Atomic Multicast

4.1   Reliable Multicast

Reliable Multicast is defined by two primitives: *R_multicast* and *R_deliver*, which enable to send and deliver messages according to the validity, integrity and agreement properties given in Section 3.1, when replacing A_multicast and A_deliver with R_multicast and R_deliver respectively. Our Reliable Multicast algorithm and the associated proofs are given in Appendix.

4.2   Weak Interactive Consistency

The WIConsistency problem is a variation of the original *Interactive Consistency* problem [5]. In the latter, each correct process proposes its initial value, and then each correct process

must decide on *the same vector of initial values* with an element for each process; the element corresponding to a given correct process must be the initial value of this process. The WIConsistency problem has a weaker specification: each correct process proposes its initial value, and processes must eventually decide on *the same set of initial values* which contains *at least one* value corresponding to the initial value of a correct process.

A correct process launches an instance of WIConsistency by invoking the *propose* primitive with its initial value. When an instance of WIConsistency is completed for some correct process p, we say that p *decide*s and the decision value is the value returned by the *propose* primitive. More precisely, the problem is characterized by the properties given below. Section 5 presents our WIConsistency protocol.

- *Agreement.* No two correct processes decide differently.
- *Validity.* The decided set contains at least one initial value of a correct process.
- *Termination.* Every correct process eventually decides.

4.3 Composing Atomic Multicast Protocol

We now describe how, in presence of Byzantine failures, the two above abstractions can be transformed into an Atomic Multicast protocol. The principle of this transformation is inspired from the structure of [2] that transforms Consensus and Reliable Broadcast into Atomic Broadcast in the crash model. The complete proofs of Atomic Multicast are given in Appendix.

Client Side. As conveyed by Algorithm 1, to A_multicast a request message m, a correct client c simply invokes $R\_multicast(m)$,

---
Algorithm 1 *Atomic Multicast Protocol: A_multicast Primitive*

A_multicast **for a correct process** c **occurs as follows:**
R_multicast(m)

Server Side. Algorithm 2 describes our implementation of the A_deliver primitive. Roughly speaking, the A_deliver primitive consists in a sequence of WIConsistency, each instance of WIConsistency being responsible for atomically delivering a batch of requests. Every correct replica in g executes concurrently $Task$ 1 and $Task$ 2 to *A_deliver* requests received from clients. These tasks manipulate three sets of messages:

- *R_delivered.* This set contains the messages delivered to replica p via primitive R_deliver. In $Task$ 1 (lines 6-7), each time a correct replica p R_delivers a message, it inserts it into *R_delivered*.
- *A_delivered.* This set contains messages that have been atomically delivered, i.e., using primitive A_deliver (line 16).
- *A_undelivered.* This set contains the messages that have been reliably delivered, i.e., R_delivered, but not yet atomically delivered.

In $Task$ 2 (lines 8-16), when a correct process p notices that the set *A_undelivered* is not empty, p launches a new instance of WIConsistency with *A_undelivered* as p's initial value,

6

and k as a sequence number that disambiguates concurrent executions of WIConsistency (line 11). Then p waits for the decision value which is the set $DecideSet$ (line 12). Having set $DecideSet$, process p collects the union of all messages that appear in this set $(\cup_i DecideSet[i])$. Then, p removes all garbage messages, i.e., messages that are either not correctly signed or messages which have the same sequence number but different content (line 13).

Removing non correctly signed messages avoids to A_deliver messages whose senders cannot be authenticated. This prevents a Byzantine process to mislead other processes by impersonating the identity of other correct processes. In addition, removing messages that have the same sequence number but different contents contributes to ensure the semantics of at most one of Atomic Multicast. Note that, such misleading messages can be safely removed because no correct process sends non correctly signed messages or different messages with identical sequence number. Finally, if some messages that are already A_delivered are present in the resulting $SetDecide$, they are also removed (line 14). Then, p A_delivers all messages that remain in $DecideSet$ according to some deterministic order (line 15), and adds them to $A\_delivered$ (line 16).

Algorithm 2 *Atomic Multicast Protocol: A_deliver Primitive*

1: **Initialization**
2: $R\_delivered \leftarrow \emptyset$
3: $A\_delivered \leftarrow \emptyset$
4: $k \leftarrow 0$
5: $A\_delivers$ **for a correct process** $p \in g$ **occurs as follows:**
6: **when** $R\_deliver(m)$ {*Task1*}
7: $\quad R\_delivered \leftarrow R\_delivered \cup \{m\}$
8: **when** $R\_delivered - A\_delivered \neq \emptyset$ {*Task2*}
9: $\quad k \leftarrow k + 1$
10: $\quad A\_undelivered \leftarrow R\_delivered - A\_delivered$
11: $\quad propose(k, A\_undelivered)$
12: $\quad$ **wait until** $decide(k, DecideSet)$
13: $\quad DecideSet \leftarrow \cup_i DecideSet[i] - \{DecideSet[i] \mid (DecideSet[i]$ *is not correctly signed*$) \vee (\exists DecideSet[j] : seq(DecideSet[i]) = seq(DecideSet[j]) \wedge DecideSet[i] \neq DecideSet[j])\}$
14: $\quad A\_deliver^k \leftarrow DecideSet - A\_delivered$
15: $\quad$ atomically deliver messages in $A\_deliver^k$ in some deterministic order
16: $\quad A\_delivered \leftarrow A\_delivered \cup A\_deliver^k$

---

5 A Weak Interactive Consistency Protocol

The full version of WIConsistency is given by Algorithm 3. This algorithm processes in asynchronous rounds and relies on the rotating coordinator and the failure detector paradigms.[5] Before proceeding with the description of the algorithm, we first define the key notion of *certificate*.

Certificates. Certificates are introduced to cope with so-called *invalid messages*. To define the notion of invalid messages, we introduce the relationship $\dot{\prec}$ between two events. Let $e1$ be the event of receiving a set of messages $sm1$ by a process p, and let $e2$ be the event of sending some message $m2$ by the same process p. We say that $e1$ *precedes* $e2$, noted $e1 \dot{\prec} e2$,

---

[5]This algorithm is inspired by a decentralized version of Consensus devised for the crash model [14].

7

if the event $e2$ of sending $m2$ is conditioned by the event $e1$ of receiving set $sm1$.[6] From an algorithmic viewpoint, this definition can be translated as follow: "if receive $sm1$ then send $m2$". In a trusted model, i.e., one that excludes malicious behaviors, when some process performs an event $e2$, such that $e1 \dot{\prec} e2$, it is trivially ensured that (1) $e1$ happened before $e2$, and (2) $sm1$ was correctly taken into account to compute $m2$. In contrast, this is no longer guaranteed in a Byzantine model. A Byzantine process may perform $e2$ either without hearing about $e1$, or without taking into account the occurrence of $e1$. The resulting message $m2$ sent by a Byzantine process is referred to as an invalid message.

The validity of some message $m2$ is proved by exhibiting, in the certificate appended to $m2$, that the reception of set $sm1$ has indeed occurred. Then, based on set $sm1$, any correct process can check if $sm1$ was correctly taken into account to compute $m2$. In other words, having set $sm1$ and knowing how $sm1$ should be taken into account to generate $m2$, each correct process can check the validity of $m2$. Therefore, the structure of any certificate consists in a collection of signed messages that compose $sm1$.[7]

Detailed Algorithm

The algorithm starts with a preliminary phase during which a set of $f+1$ values collected from different processes is constructed. Then, after this phase, two concurrent tasks are launched ($Task$ 1 and $Task$ 2) to allow correct processes to eventually decide on the same set of values. In the following, p is any correct process executing Algorithm 3, while q is another process (correct or not) with which p interacts.

Preliminary Phase. (lines 3-6) During this phase, process p sends all processes its initial value in a signed message. Then, p collects $f+1$ correctly signed values from different processes. This set of values set_p is the **estimate** with which p participates in $Task1$.

Task 1. Task 1 is divided in two phases. In Phase 1, every correct process tries to decide on the estimate of the current coordinator c_p (see Phase 1 of Figure 2). If c_p is suspected by at least $2f+1$ processes, then p proceeds to Phase 2 before moving to the next round (see Phase2 of Figure 2). Throughout Task 1, a local predicate $Byzantine_p(q)$ is associated to every process q by p; initially, this predicate is *false*. As soon as p detects some misbehavior exhibited by q, like sending invalid messages or sending an estimate that is not a set of values, p sets its local predicate $Byzantine_p(q)$ to true, which means that p suspects q.

- Phase 1 (lines 10-27). During Phase 1, current coordinator c_p uses a centralized Echo Broadcast [16, 12] to send its estimate to all processes. The Echo Broadcast protocol prevents a Byzantine coordinator from sending different estimates to different correct processes; this is an instance of the well-known Byzantine Generals Problem [9]. So, first c_p sends its estimate set_{c_p} in a signed *Initial* message to all processes (line 11). When process p receives this message for the first time, p checks if it is a valid message, i.e., with an estimate composed of $f+1$ values, and if so, it sends an *Echo* message to c_p (lines 13-15). Finally, once c_p collected $2f+1$ *Echo* messages, it sends a *Ready*

---

[6]Note that the precedence relationship used here is slightly different from that of [10], in that it involves a message and a *set of messages* (rather than two messages).

[7]At the beginning of the protocol, i.e., $r=1$, some messages $m2$ may not carry any certificate because there is no previous $sm1$ was received. In such a case, $m2$ is validated by an empty certificate.

Algorithm 3 Byzantine-Resilient WIConsistency Algorithm

1: **function** *propose(e_p)*   {*Every process p executes Task 1 and Task 2 concurrently*}
2: $InitialCertif_p \leftarrow \emptyset$
3: $set_p \leftarrow \emptyset$ {*Preliminary Phase*}
4: send (p, e_p) to all
5: **wait until** $(f + 1 \; processes \; q : received \; (q, e_q))$
6:   $set_p \leftarrow \{(q, e_q) \mid p \; received \; (q, e_q) \; from \; q\}$
7: **loop**   { *Task 1*}
8:   $CurrentRoundTerminated_p \leftarrow false; \; ReadyCertif_p \leftarrow \emptyset; \; GoPhase2Certif_p \leftarrow \emptyset$
9:   $DecideCertif_p \leftarrow \emptyset; \; coordSuspect_p \leftarrow false; \; c_p \leftarrow (r_p \; mod \; N) + 1; \; phase_p \leftarrow 1$
10:   **if** $c_p = p$ **then**
11:     send $(Initial, p, r_p, set_p, InitialCertif_p)$ to all

12:   **while** not $(CurrentRoundTerminated_p)$ **do**
13:     **when** receive Valid $(Initial, q, set_q, InitialCertif_q)$ from q
14:       **if** $(q = c_p) \wedge (no \; Echo \; message \; was \; sent \; in \; r_p \; by \; p)$ **then**
15:         send $(Echo, p, r_p, set_q)$ to c_p

16:     **when** $(c_p = p) \wedge (for \; 2f + 1 \; distinct \; processes \; q: \; received \; (Echo, q, r_p, set_p))$
17:       $ReadyCertif_p \leftarrow \{(Echo, q, r_p, set_p) \mid p \; received \; (Echo, q, r_p, set_p) from \; q\}$
18:       send $(Ready, p, r_p, set_p, ReadyCertif_p)$ to all

19:     **when** receive Valid $(Ready, q, r_p, set_q, ReadyCertif_q)$
20:       $DecideCertif_p \leftarrow DecideCertif_p \cup \{(Ready, q, r_p, set_q, ReadyCertif_q)$
21:       **if** (first *Ready* message received) $\wedge (p \neq c_p)$ **then**
22:         $ReadyCertif_p \leftarrow ReadyCertif_q$
23:         $set_p \leftarrow set_q$
24:         send $(Ready, p, r_p, set_p, ReadyCertif_p)$ to all
25:       **else if** $2f + 1$ *Ready* messages received from distinct processes **then**
26:         send $(Decide, p, r_p, set_p, DecideCertif_p)$ to all
27:         $return(set_p)$

28:     **when** $(c_p \in \Diamond M_{\mathcal{A}} \vee Byzantine_p(c_p)) \wedge (not \; coordSuspected_p)$
29:       send $(Supicion, p, r_p)$ to all
30:       $coordSuspected_p \leftarrow true$

31:     **when** $(phase_p = 1) \wedge (for \; 2f + 1 \; distinct \; processes \; q: \; received \; (Supicion, q, r_p))$
32:       $GoPhase2Certif_p \leftarrow \{(Supicion, q, r_p) \mid p \; received \; (Supicion, q, r_p) \; from \; q\}$
33:       send $((GoPhase2, p, r_p, set_p, ReadyCertif_p), (GoPhase2Certif_p))$ to all
34:       $phase_p \leftarrow 2; \; InitialCertif_p \leftarrow \emptyset$

35:     **when** receive Valid $((GoPhase2, q, r_p, set_q, ReadyCertif_q), (GoPhase2Certif_q))$
36:       **if** $phase_p = 1$ **then**
37:         $phase_p \leftarrow 2; \; InitialCertif_p \leftarrow \emptyset$
38:         send $((GoPhase2, p, r_p, set_p, ReadyCertif_p), (GoPhase2Certif_q))$ to all
39:       $InitialCertif_p \leftarrow InitialCertif_p \cup (GoPhase2, q, r_p, set_q, ReadyCertif_q)$
40:       **if** $2f + 1$ *GoPhase2* messages received from distinct processes **then**
41:         $(set_p, ReadyCertif_p) \leftarrow Last\_Successfully\_EchoCertified(InitialCertif_p)$
42:         $currentRoundTerminated_p \leftarrow true$
43:         $r_p \leftarrow r_p + 1$

44: **when** receive Valid $(Decide, q, r, set, DecideCertif_q)$ {*Task2*}
45:   send $(Decide, q, r, set, DecideCertif_q)$ to all
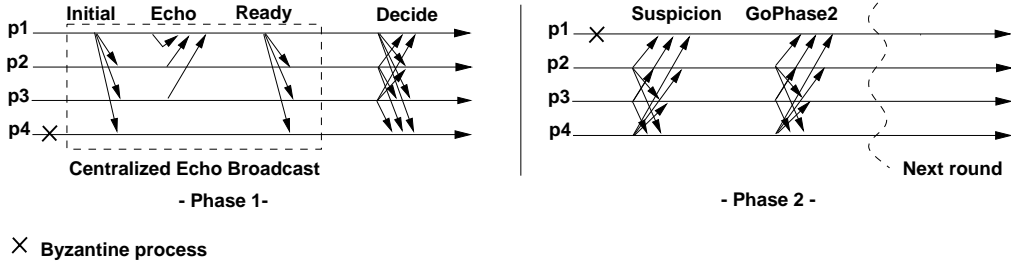46: $return(set)$

9

Figure 2: Task 1 of Weak Interactive Consistency

message to all processes (lines 16-18). At this point the Echo Broadcast protocol is completed. When c_p sends a valid *Ready* message, we say that c_p has *successfully echo certified* its estimate. Then, when process p receives a valid *Ready* message for the first time, it adopts set_{c_p} and relays the *Ready* message to all processes (lines 19-24). Once p received $2f + 1$ *Ready* messages containing c_p's estimate, p sends a *decide* message to all processes and decides on set_p (line 25-27).

- Phase 2 (lines 28-43). Phase 2 ensures that if any process decides on some estimate *set* during Phase 1 of round r, then every correct process that starts round $r + 1$ starts it with its estimate equal to *set*. To proceed to Phase 2, a correct process must learn that at least $2f + 1$ processes have suspected the current coordinator (lines 28-30). So, once $2f + 1$ *Suspicion* messages were received by p, a *GoPhase2* message is sent to all (lines 31-33). This message carries the last valid estimate delivered by p, i.e., the last successfully echo certified estimate delivered by p. When the reception of a valid *GoPhase2* occurs at p, if p is still in Phase 1, it proceeds to Phase 2. Then p waits for the reception of $2f + 1$ valid *GoPhase2* messages (lines 34-40). Among the $2f + 1$ valid *GoPhase2* messages received, process p looks for the last estimate that was successfully echo certified (line 41). If such value exists, p adopts it, otherwise it does not updates its estimate. Then p proceeds to the next round (lines 42-43).

Task 2 (lines 44-46). This task handles the reception of *Decide* messages by process p. If the *Decide* message is valid, then first p relays it to all processes, and second it decides on the value carried by the *Decide* message.

6   Safety & Liveness of Weak Interactive Consistency

The WIConsistency is at the heart of the safety and liveness of our state machine replication protocol. For this reason, this section sketches how these two properties are ensured in the WIConsistency protocol proposed in Section 5. The formal proofs of all the Protocols used in building the modular machine replication protocol are given in Appendix.

Safety. The safety of WIConsistency encompasses both agreement and validity. Agreement relies on the Echo Broadcast protocol, on the locking of the decision value, and on certificates. The Echo Broadcast protocol preserves agreement within a round. When a decision is made on some estimate *set*, we have at least $f + 1$ correct processes that delivered *set*. Once *set*

10

is delivered by $f + 1$ correct processes, there is no other estimate that can be delivered by correct processes from that time onward; this is what we mean by "locking" the decision value. Certificates prevent Byzantine processes from misleading correct ones into deciding on different estimates, which would result in violating the agreement property.

Regarding the validity property, no process can successfully echo broadcast an estimate that does not match the expected format, i.e., not composed of $f+1$ correctly signed messages (q, e_q). So, any decided estimate is a set of $f+1$ values. In the presence of at most f Byzantine processes, this set contains at least one initial value of a correct process. In conclusion, notice that the safety properties (agreement and validity) are preserved whatever the assumptions on system synchrony, i.e., the outputs of failure detector.

Liveness. The liveness of WIConsistency corresponds to its termination property. The correctness of this property is based on the presence of at least $2f + 1$ correct processes, on reliable and FIFO communication channels, and on the properties of failure detector $\diamond M_{\mathcal{A}}$. The presence of at least $2f + 1$ correct processes and of reliable communication channels ensures that when $2f + 1$ messages are expected, they are eventually received, thus avoiding any blocking *wait* in our algorithm. The Mute $\mathcal{A}$-completeness of $\diamond M_{\mathcal{A}}$ prevents Byzantine processes from stopping the progress of the protocol, by suspecting mute processes. Indeed, the malicious processes that can prevent the progress of the WIConsistency protocol are those that crash or arbitrarily decide to stop sending messages.

Furthermore, Eventual weak $\mathcal{A}$-accuracy expresses the ability of $\diamond M_{\mathcal{A}}$ to eventually stop falsely suspecting some correct process. This property prevents correct processes from always moving to the next round without deciding, i.e., it allows them to reach some round with no suspected correct coordinator; the protocol can then terminate. Note that a Byzantine process could stop sending expected messages but continue sending other messages, like *Suspicion* messages (line 29). For our failure detector, such a process is not mute. Furthermore, some messages sent by such a process could be always be valid since they do not need a certificate.[8] We solve this problem by using the FIFO property of our communication channels and by using the fact that, in any round, every correct process only sends a bounded number nb of messages. Consequently, if a correct process p expects in round r a message from some process q, it periodically checks if it receives more than $nb * r$ messages from q without receiving the expected message. With this test, Byzantine processes that skip expected messages without being mute are detected.

7 Concluding Remarks

In this paper, we advocate the divide-&-conquer approach for reducing the complexity of Byzantine-resilient replication protocols. More precisely, we proposed a modular approach for devising a state machine replication protocol, based on a set of well-defined abstractions. Our modular approach allowed us to identify the synchrony assumptions under which the liveness of our protocol is ensured; as for safety, it is always ensured in presence of less than one third of Byzantine processes. Synchrony assumptions are encapsulated in a Byzantine failure detector $\diamond M_{\mathcal{A}}$, which differs from the original definition of [2] in that its specification is not orthogonal to the algorithm using it. More precisely, $\diamond M_{\mathcal{A}}$ captures processes that stop sending *algorithmic* messages. In a Byzantine model, one cannot avoid the dependency

---

[8]Not all messages in our algorithm need certificate to be valid, e.g., *Suspicion* messages.

between a failure detector and the algorithm using it, because incorrect processes exhibiting a crash-like behavior are not only those that really crash, but may also be processes that arbitrary decide to stop sending algorithmic messages. All Byzantine failure detectors specifications we know of are, in one way or another, linked to the algorithm that uses them [11, 8]. Interestingly, even in [1] where no failure detector is used, timeouts are set to detect the non-reception of algorithmic messages.

Regarding performance issues, some optimizations can be introduced in our protocol to make it more efficient in failure-free runs without suspicions. First, to send its requests, a client can aim only at the first coordinator via a point-to-point communication, rather than using a Reliable Multicast. If the client suspects some misbehavior, e.g., if it gets no reply after some time, a Reliable Multicast can then be used to send the request to all replicas. Second, rather than deciding on a set of values, the replicas can start by deciding only on one value. That is, initially a simple Consensus is used rather than WIConsistency. Again, if some client suspects that its request is not being treated, WIConsistency can be launched. With such optimizations and with the use of a decentralized Echo Broadcast, our protocol becomes as efficient as the monolithic protocol of [1], without compromising modularity.

References

[1] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, New Orleans, USA, February 1999.

[2] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[3] A. Doudou, B. Garbinato, R. Guerraoui, and A. Schiper. Muteness Failure Detectors: Specification and Implemention. In *European Dependable Computing Conference*, volume 1667 of *LNCS*. Springer Verlag, September 1999.

[4] M. Fischer, N. Lynch, and M. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32:374–382, April 1985.

[5] M. J. Fischer. The consensus problem in unreliable distributed systems (A brief survey). In *Proc. Int. Conf. on Foundations of Computations Theory*, pages 127–140, Borgholm, Sweden, 1983.

[6] V. Hadzilacos and S. Toueg. Fault-Tolerant Broadcasts and Related Problems. In Sape Mullender, editor, *Distributed Systems*, pages 97–145. ACM Press, 1993.

[7] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. The secure protocols for securing group communication. In *Proceedings of the 31st Hawaii International Conference on System Sciences*, volume 3, pages 317–326. IEEE, January 1998.

[8] K. P. Kihlstrom, Louise E. Moser, and P. M. Melliar-Smith. Solving consensus in a Byzantine environment using an unreliable fault detector. In *Proceedings of the International Conference on Principles of Distributed Systems (OPODIS)*, pages 61–75, December 1997.

[9] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[10] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[11] D. Malkhi and M. Reiter. Unreliable Intrusion Detection in Distributed Computations. In *Proc. 10th Computer Security Foundations Workshop (CSFW97)*, pages 116–124, June 1997.

[12] M.K. Reiter. Secure Agreement Protocols: Reliable and Atomic Group Multicast in Rampart. In *Proc. 2nd ACM Conf. on Computer and Communications Security*, pages 68–80, November 1994.

[13] R.L. Rivest, A. Shamir, and L.Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[14] A. Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3):149–157, April 1997.

[15] F. B. Schneider. Replication Management Using the State-Machine Approach. In Sape Mullender, editor, *Distributed Systems*, pages 169–197. ACM Press, 1993.

[16] S. Toueg. Randomized Byzantine Agreements. In *Proc. of the 3rd ACM Symposium on Principles of Distributed Computing*, pages 163–178, August 1983.

Appendix

Reliable Multicast Protocol & Correctness Proofs

Algorithms 4 and 5 give a simple implementation of Reliable Multicast based on a diffusion mechanism. We then prove that these algorithms satisfy Reliable Multicast properties.

Algorithm 4 *Reliable Multicast: R_multicast Primitive*

    R_multicast **for a correct process c occurs as follows:**
    send(m) to all processes in g

As conveyed by Algorithm 5, the *R_deliver* primitive, performed by correct process, is based on a diffusion mechanism. That is, every correct process p in g that receives for the first time a correctly signed message m relays m to all processes in g.

Algorithm 5 *Reliable Multicast: R_deliver Primitive*

    R_deliver **for every correct process $p \in g$ occurs as follows:**
    when receive (m)
      if (m is correctly signed) $\wedge$ (first reception of m) **then**
        send m to all processes in g
        R_deliver (m)

Correctness Proofs

Theorem 7.1 *Algorithms 4 and 5 satisfy the Reliable Multicast properties.*

Proof: We consider the three properties separately.

- *Validity:(If a correct process R_multicasts a message m, then some correct process in g eventually R_delivers m.)*
  By assumption, a correct process c R_multicasts only correctly signed messages m. From Algorithm 4, an R_multicast aims at sending m to all processes in g. By the reliability property of the channels, some correct process $p \in g$ eventually receives the correctly signed message m and hence R_delivers it.

- *Agreement: (If a message m is R_delivered by some correct process in g, then all correct processes in g eventually R_deliver m.)*
  Let p and q be any two correct processes in g, such that p R_delivers message m. We must show that q also eventually R_delivers m. By Algorithm 5, if p R_delivers m, then m is a correctly signed message that p must have relay to all other processes in g. Therefore, by the reliability property of the channels, process q eventually receives m and, being correct, R_delivers m.

- *Integrity: (For any message m, every correct process p in g A_delivers m at most once. Furthermore, if the sender of m, say q, is correct, then no correct process in g R_delivers m unless q has previously R_multicast m)*
  From Algorithm 5, any correct process $p \in g$ R_delivers m only if this is the first reception of m. Furthermore, we must show that if p R_delivers m, then if the sender of m is a correct process q, then q has in fact R_multicasts m. From Algorithm 5,

14

process p R_delivers m only if m is correctly signed by some process q in the system. If q is correct, then by the message unforgeability property, q is the only process that originates m. Being correct, q has performed an R_multicast of m.

$\square$

Correctness Proofs of Atomic Multicast Protocol

Here we prove that Algorithms 1 and 2 satisfy Atomic Multicast properties. In the following, when we refer to some variable, say v, that belongs to process p, we use the notation v_p (e.g., $R\_delivered_p$).

Lemma 7.2 *Consider any two correct processes p and q both in g, and any message m. If $m \in R\_delivered_p$, then eventually $m \in R\_delivered_q$.*

Proof: Trivially follows from the agreement property of Reliable Multicast. $\square$

Lemma 7.3 *For any two correct processes p and q both in g, and all $k \geq 1$:*

1) *If p executes $propose(k, *)$, then q eventually executes $propose(k, *)$.*

2) *If p A_delivers messages in $A\_deliver_p^k$, then q eventually A_delivers messages in $A\_deliver_q^k$, and $A\_deliver_p^k = A\_deliver_q^k$.*

Proof: The proof is by simultaneous induction on 1) and 2).

Lemma part 1, $k = 1$: We prove that if $p \in g$ executes $propose(1, *)$ then $q \in g$ will eventually execute $propose(1, *)$. Since $A\_delivered_p$ is initially empty, $R\_delivered_p$ must contain some message m, when p executes $propose(1, *)$. Then by Lemma 7.2, m is eventually in $R\_delivered_q$ for every correct process $q \in g$. Since $A\_delivered_q$ is initially empty for every correct process q, $R\_delivered_q - A\_delivered_q \neq \emptyset$ eventually holds. So, every process q eventually executes $propose(1, *)$.

Lemma part 2, $k = 1$: We prove that if p A_delivers messages in $A\_deliver_p^1$, then q eventually A_delivers messages in $A\_deliver_q^1$ such that $A\_deliver_p^1 = A\_deliver_q^1$. If p A_delivers messages in $A\_deliver_p^1$, then it has previously executed $propose(1, *)$. By part 1 of lemma 7.3, every correct process in g eventually executes $propose(1, *)$. By termination, validity and agreement properties of WIConsistency, all correct processes in g eventually decide on the same set of messages $DecideSet^1$. That is $DecideSet_p^1 = DecideSet_q^1$. Let $GarbageSet_p^1$ be the subset (possibility empty) of $DecideSet_p^1$ such that all messages in $GarbageSet_p^1$ are either not correctly signed or/and composed of different contents and similar sequence number. By assumption, every correct process knows the identities (public keys) of all other processes (we remind that there is a static group of processes). Furthermore, relying on the public key diffusion mechanism, we assume that all correct processes obtain the same public key for every process in the system. As a consequence, if some message m in $SetDecide^1$ is considered by some correct process $p \in g$ as a no correctly signed message, then every correct process $q \in g$ consider m as a no correctly signed message. Hence $GarbageSet_p^1 = GarbageSet_q^1$ for any two correct process p and q in g. Therefore, removing $GarbageSet^1$ from $DecideSet^1$ results in the same new $DecideSet^1$ for all correct processes. Since $A\_delivered_p^1$ and $A\_delivered_q^1$ are initially empty, and $DecideSet_p^1 = DecideSet_q^1$ then $A\_deliver_p^1 = A\_deliver_q^1$.

15

Lemma part 1, $k = n$**:** We assume that the lemma holds for all k such that $1 \leq k < n$. We prove that if $p \in g$ executes $propose(n, *)$, then eventually $q \in g$ executes $propose(n, *)$. If p executes $propose(n, *)$, then there is some message $m \in R\_delivered_p$ and $m \notin A\_delivered_p$. Thus, m is not in $\cup_{k=1}^{n-1} A\_deliver_p^k$. By, the induction hypothesis, for all k, we have $A\_deliver_p^k = A\_deliver_q^k$. So, m is not in $\cup_{k=1}^{k=n-1} A\_deliver_q^k$. Since m is in $R\_delivered_p$, by lemma 7.2, m is eventually in $R\_delivered_q$. Therefore, for every process q $R\_delivered_q - A\_delivered_q \neq \emptyset$ eventually holds. So, each process q eventually executes $propose(n, *)$.

Lemma part 2, $k = n$**:** We prove that if $p \in g$ A_delivers messages in $A\_deliver_p^n$, then $q \in g$ eventually A_delivers messages in $A\_deliver_q^n$ such that $A\_deliver_p^n = A\_deliver_q^n$. If p A_delivers messages in $A\_deliver_p^n$, then p has previously executed $propose(n, *)$. By part 1 of this lemma, eventually each correct process in g executes $propose(n, *)$. By the termination, validity and agreement properties of WIConsistency, all correct processes in g eventually decide on the same set of messages $DecideSet^n$. Consequently, we have $DecideSet_p^n = DecideSet_q^n$. For similar reasons to those stated in part 2 of Lemma 7.3, removing $GarbageMsg^n$ from $DecideSet^n$ results in the same new $DecideSet^n$ for all correct processes. Note that, $A\_deliver_p^n = SetDecide^n - \cup_{k=1}^{k=n-1} A\_deliver^k$, and $A\_deliver_q^n = SetDecide^n - \cup_{k=1}^{k=n-1} A\_deliver^k$. By the induction hypothesis for all $1 \leq k < n$, we have $A\_deliver_p^k = A\_deliver_q^k$. Since in addition $SetDecide_p^n = SetDecideq_n$, then $A\_deliver_p^n = A\_deliver_q^n$.

$\square$

Theorem 7.4 *Algorithms 1 and 2 satisfy the agreement and total order properties of Atomic Multicast (Agreement and Total Order).*

Proof: From Lemma 7.3, all correct processes in g decide on the same batch of messages. Then, the same deterministic order is followed by all correct processes in g to deliver messages in the decided batch. Thus, the agreement and total order are ensured.

$\square$

Theorem 7.5 *If a correct process A_multicasts a message m, then some correct process in g eventually A_delivers m (Validity).*

Proof: The proof is by contradiction. Assume that a correct process q A_multicasts a message m, and some correct process $p \in g$ never A_delivers m. By Lemma 7.4, we know that if some correct process $p \in g$ does not A_deliver m, then no correct process $q \in g$ A_delivers m.

By Algorithm 2, we know that, to A_multicast a message m, a correct process executes a R_multicast of m (m is previously correctly signed by q). Therefore, thanks to validity and agreement properties of Reliable Multicast, eventually every correct process $p \in g$ R_delivers m in Task 1, i.e., eventually $m \in R\_delivered_p$. On the other side, since no correct process $p \in g$ never A_delivers m, so no process p inserts m in $A\_delivered_p$.

From Algorithm 2, there exists a constant k, such that for all $l \geq k$, every process p has $m \in R\_delivered_p^l - A\_delivered_p^l$, i.e., $m \in A\_undelivered_p^l$. Then, every process p launches its lth instance of WIConsistency, with $A\_undelivered_p^l$ as the proposed value. By termination and agreement of WIConsistency, we infer that every correct process eventually decides on the same $DecideSet^l$. By the validity property of WIConsistency, we know that

16

$DecideSet^l$ contains the proposed value of at least one correct process s, i.e, $A\_undelivered^l_s$. Therefore, we have $m \in DecideSet^l$. Since m is sent by a correct process, then m is a correctly signed message and q being correct associates one sequence number per message. Therefore, no correct process removes m from $DecideSet^l$. Since the decision is on the union of all messages in $DecideSet^l$, except garbage ones and those which are already $A\_deliver$ed, then m is eventually $A\_deliver$ed by all correct processes: this contradicts our initial assumption. □

Theorem 7.6 *For any message m, every correct process p in g A_delivers m at most once. Furthermore, if the sender of m, say q, is correct, then no correct process in g A_delivers m unless q has previously A_multicast m (Integrity).*

Proof: Every message is identified by its sequence number that should be unique. In some stage k of Atomic Multicast, all messages that have similar sequence number but different contents are removed from the set to A_deliver. Therefore, in some stage k a message m is A_delivered at most once. Moreover, when p A_delivers a message m, then p inserts m in its set of delivered messages $A\_delivered_p$. So, from Algorithm 2, process p delivers m at most once.

Now, we have to prove the second clause of the integrity property. If a correct process p A_delivers a message m at some stage k of Atomic Multicast, then it has previously decided on $DecideSet^k$. Therefore, m was proposed by some process s during the kth instance of WIConsistency. Here, we distinguish two cases:

- Process s is correct: To propose m process s should have previously R_deliver m. From the integrity property of Reliable Multicast, if the sender of m is a correct process q, then q has R_multicast m. Being correct q has A_multicast m.

- Process s is Byzantine: In this case, since s is a Byzantine process we cannot infer that s has R_deliver m. However, process p does not deliver m if m is not a correctly signed message. Thus, if p A_delivers m, then p has authenticated the sender q of m. Therefore, from the message unforgeability property, if process q is correct no process except q has sent m. Since q is a correct, then q has R_multicast m and hence has A_multicast m. □

Correctness Proofs of WIConsistency Protocol

Here we prove that Algorithm 3 solves WIConsistency in presence of Byzantine failures. The correctness proofs aim at showing that Algorithm 3 preserves the agreement, validity and termination properties. For simplicity, we assume that $N = 3f + 1$, which does not contradict the assumption of at most $f < N/3$ Byzantine processes among the N servers.

Lemma 7.7 *In any round r, the coordinator (correct or not) of r cannot successfully echo certify two different estimates $set1$ and $set2$.*

PROOF: The proof is by contradiction.

Assume that in round r the associated coordinator, say process p, successfully echo certify two estimates $set1$ and $set2$. Consequently, process p constructs: (1) a *Ready* message for $set1$, with a $ReadyCertif$ composed of $2f+1$ messages $(Echo, p_j, r, set1)$, and (2) a *Ready* message

for $set2$, with $ReadyCertif$ composed of $2f + 1$ messages $(Echo, p_j, r, set2)$. Therefore, in round r, two sets of $2f + 1$ processes sent $Echo$ messages for two distinct estimates $set1$ and $set2$. With $N = 3f + 1$ and at most f Byzantine processes, the previous statement is correct only if at least one correct process sent two $Echo$ messages one for $set1$ and one for $set2$. Since each correct process sends at most one $Echo$ message then, this statement cannot be true. So, p cannot construct two valid $ReadyCertif$ in the same round r for two different estimates $set1$ and $set2$. As a consequence, no two different estimates can be successfully echo certified in the same round: a contradiction. □

Lemma 7.8 *Let r be the smallest round for which at least $f + 1$ correct processes delivered an estimate $set1$ that was successfully echo certified. Then, no other value $set2 \neq set1$ can be successfully echo certified in any subsequent round r' (i.e., $r' \geq r$).*

PROOF: The proof is split in two cases.

Case 1 $(r = r')$ **:** Immediate from Lemma 7.7.

Case 2 $(r' > r)$**:** The proof is by contradiction. Assume that l is the first round for which the lemma does not hold: in round l there is at least one correct process q that delivers an estimate $set2 \neq set1$.

So, in round l, q has received $set2$ in a valid $Ready$ message, i.e., a $Ready$ with a $ReadyCertif$ that is composed of $2f + 1$ signed $Echo$ messages $(Echo, p_i, l, set2)$. Thus there is at least $f+1$ correct processes that sent an $Echo$ message for $set2$ to p_l the coordinator of round l. That means these correct processes have received a valid $Initial$ message from p_l. That is, they received an $Initial$ message with a certificate $InitialCertif$ composed of $2f+1$ signed $GoPhase2$ messages $(GoPhase2, p_i, l-1, set_i, ReadyCertif_i)$ such that the estimate set_i with the last $ReadyCertif$ corresponds to $set2$ or all $ReadyCertif$ are empty. Let $ProcessSet1$ be such a set of $2f + 1$ processes that send their $GoPhase2$ messages to p_l in round $l - 1$.

By assumption, we know that the lemma holds for all rounds k such that $r \leq k \leq l-1$. Hence, until round $l - 1$ no estimate $set2 \neq set1$ was successfully echo certified. In addition, we know that there is at least $f + 1$ correct processes in the system that delivered $set1$. Let $ProcessSet2$ be this set of correct processes. With $N = 3f + 1$, we have $|ProcessSet1| \cap |ProcessSet2| \neq \emptyset$. Therefore, there is at least one correct process s that belongs to both $ProcessSet1$ and $ProcessSet2$. So, some correct process s participates to the construction of $CertifInitial$ of process p_l. Being correct, process s sends to p_l a $GoPhase2$ message with its current valid estimate, i.e., $set1$. Since in all rounds k such that $r \leq k \leq l-1$, there is not an other value that was successfully echo certified, then $set1$ is the last one. Thus, p_l cannot construct a valid $Initial$ message for any estimate $set2 \neq set1$: contradiction.

□

Theorem 7.9 *No two correct processes decide differently (Agreement).*

PROOF: The proof is by contradiction. Assume that two correct processes, say p and q, decide on two different estimate, $set1$ and $set2$ respectively, during rounds r and r' such that $r \leq r'$ (the same proof holds if $r' \leq r$). To decide each correct process must receive at least $2f + 1$

Ready messages for the same estimate (if it either decides in Task 1 or Task 2). Therefore, when p decides on estimate $set1$, in round r , there is at least $f + 1$ correct processes that deliver the same value $set1$ in some round $k \leq r$. Consequently, from Lemma 7.8, there is no other value $set2$ different than $set1$ that could be successfully echo certified in any round $r' \geq k$. Consequently, the correct process q could not decide on some $set2 \neq set1$. This contradicts our assumption, i.e., q decides $set2 \neq set1$. □

Lemma 7.10 *If one correct process decides, then every correct process eventually decides.*

PROOF: Let p be a correct process that decides either in Task 1 or Task 2. In both cases, p sends a valid *Decide* message to all before deciding. By the assumption of reliable channels, every correct process that has not yet decided eventually receives the valid *Decide* message, which is handled by Task 2, and then decides. □

Lemma 7.11 *With a failure detector $\Diamond M_{\mathcal{A}}$ that satisfies Mute Completeness, Algorithm 3 ensures that if no correct process decides in round r eventually all correct processes proceed to round $r + 1$.*

PROOF: The proof is by contradiction. Let r be the smallest round for which the lemma does not hold: no correct process decides in round r, and some correct process never reaches round $r + 1$. As r is the smallest of such rounds, each correct process eventually reaches round r. We show the contradiction by proving the following successive results:

1) At least one correct process eventually reaches Phase 2 of round r.

2) Each correct process eventually reaches Phase 2 of round r.

3) Each correct process eventually reaches Phase 1 of round $r + 1$.

Proof of 1): Assume that no correct process decides in round r, and no correct process reaches Phase 2 of round r. We consider two cases: the coordinator p_c of round r (a) sends at least one valid *Ready* message to some correct process, or (b) does not send any valid *Ready* message to any correct process.

Case (a): process p_c sends at least one valid *Ready* message to some correct process p. So, p receives this valid *Ready* message and relays it to all. Each correct process q is by assumption in Phase 1. So, q delivers the valid *Ready* message received from p and then, reissues the message to all. By the assumption of reliable channels and $f < N/3$, all correct processes eventually receive $2f + 1$ valid *Ready* messages. Thus eventually some correct process receives at least $2f + 1$ messages *Ready* and then decides in round r: a contradiction with the fact that no correct process decides in round r. So only case (b) remains to be considered.

Case (b): process p_c does not send any valid *Ready* message to any correct process p. This means that either (1) p_c is mute to p, and by the mute completeness of $\Diamond M_{\mathcal{A}}$ process p eventually suspects p_c, or (2) p_c sends a non valid *Ready* message or a non valid estimate to p, or p_c skips the *Ready* message, which leads in both cases the predicate $Byzantine_i(p_c)$ to become true. In any case, each correct process eventually suspects p_c and sends a *Suspicion* message to all. Thus, at least one correct process p receives $2f + 1$ *Suspicion* messages. Then, p sends a *GoPhase2* message and proceeds to Phase 2: contradiction with the fact that no correct process reaches Phase 2 of round r. □

Proof of 2): By (1), at least one correct process, say p, eventually reaches Phase 2 of round r, after sending to all a valid *GoPhase*2. Every correct process still in Phase 1 receives this message and, in turn, reaches Phase 2.

Proof of 3): By (2), all correct processes reach Phase 2 of round r. Then each correct process sends a valid *GoPhase*2 message to all. Thus each correct process receives at least $2f + 1$ messages *GoPhase*2. This allows Phase 2 of round r to be completed. Consequently, each correct process proceeds to next round, i.e., $r + 1$. □

Theorem 7.12 *With a failure detector $\diamond M_\mathcal{A}$ that satisfies Weak Accuracy, every correct process eventually decides (Termination).*

PROOF: By the eventual weak accuracy property of $\diamond M_\mathcal{A}$, there is a time t after which some correct process p is not suspected by any correct process to be mute. Let r be a round such that (1) p is the coordinator of r, and (2) every correct process enters round r after t (if such a round does not exist, then by Lemma 7.10, one correct process has decided in a round $r' < r$, and so, by Lemma 7.10 every correct process decides, and the termination property holds). As $f < N/3$ and no correct process suspects p in round r. Thus no correct process ever proceeds to Phase 2 of round r. Process p sends its valid *Initial* message. As p is correct, each correct process eventually receives and delivers its *Initial* message and then sends an *Echo* for the received message. So, p eventually receives $2f + 1$ messages *Echo* message needed to successfully echo certified its estimate. By the reliable channel, each correct process receives a valid *Ready* message and then relays it to all. Therefore, every correct process will receive enough valid *Ready* messages to decide. □

Theorem 7.13 *The decided set contains at least one initial value proposed by some correct process (Validity).*

PROOF: By the Echo Broadcast protocol introduced in Algorithm 3, no coordinator (correct or not) can construct a valid Ready message for an estimate, which is not a set of $f + 1$ signed messages (p, e_p). Since any decided estimate should be carried by a valid *Ready* message, then any decided value is a set of $f + 1$ signed messages (p, e_p). In presence of at most f Byzantine processes in any decided value there is at least one initial value of a correct process. □