Muteness Failure Detectors: Specification and Implementation*

Assia Doudou[1], Benoit Garbinato[2], Rachid Guerraoui[1**], and André Schiper[1]

[1] École Polytechnique Fédérale, Lausanne (Switzerland)
{Doudou, Guerraoui, Schiper}@epfl.ch
[2] United Bank of Switzerland, Zürich (Switzerland)
Garbinato@ubs.com

Abstract. This paper extends the failures detector approach from crash-stop failures to muteness failures. Muteness failures are malicious failures in which a process stops sending algorithm messages, but might continue to send other messages, e.g., *"I-am-alive"* messages. The paper presents both the specification of a muteness failure detector, denoted by $\diamond M_{\mathcal{A}}$, and an implementation of $\diamond M_{\mathcal{A}}$ in a partial synchrony model (there are bounds on message latency and clock skew, but these bounds are unknown and hold only after some point that is itself unknown). We show that, modulo a simple modification, a consensus algorithm that has been designed in a crash-stop model with $\diamond\mathcal{S}$, can be reused in the presence of muteness failures simply by replacing $\diamond M_{\mathcal{A}}$ with $\diamond\mathcal{S}$.

1 Introduction

A fundamental characteristic of distributed systems is the notion of partial failures: part of a system might have failed while the rest might be correct. Coping with partial failures usually requires to get some (even approximative) knowledge about which processes have failed and which have not.

1.1 Background: Crash Detectors

Traditionally, failure detection mechanisms were usually mixed up with the distributed protocols using them. Relatively recently, Chandra and Toueg suggested an approach where failure detection is encapsulated within a so-called *failure detector* and decoupled from the rest of the distributed protocol [3]. Roughly speaking, a failure detector is a distributed oracle that provides hints about partial failures in the system; a process p typically consults its failure detector module to know whether a given process q has crashed or not. Formally, a failure detector is described according to axiomatic *completeness* and *accuracy* properties. Completeness expresses the ability of a failure detector to eventually detect

* Research supported by OFES under contract number 95.0830, as part of the ESPRIT BROADCAST-WG (number 22455).

** Rachid Guerraoui is currently a Faculty Hire at HP Labs, Palo Alto (CA).

failures, while accuracy expresses its ability to eventually avoid false suspicions (i.e., falsely suspecting correct processes). By decoupling failure detection from other algorithmic issues, Chandra and Toueg provided an abstraction that avoids focusing on operational features of a given model, such as the asynchrony of the transmission and the relative speeds of processes. Features related to failure detection are encapsulated within a separate module: the failure detector becomes a powerful abstraction that simplifies the task of designing distributed algorithms[1] and proving their correctness. For example, Chandra and Toueg have described an algorithm that solves the consensus problem in an asynchronous system augmented with an *eventually strong* failure detector, denoted by $\diamond\mathcal{S}$. The axiomatic properties of $\diamond\mathcal{S}$ encapsulate the amount of synchrony needed to solve the consensus problem in the presence of crash failures and hence to circumvent the well-known *FLP* impossibility result.[2] Other agreement algorithms, e.g., [15], have later been designed on top of the same failure detector $\diamond\mathcal{S}$. Although $\diamond\mathcal{S}$ cannot be implemented in a purely asynchronous system (this would contradict the FLP impossibility result), when assuming practical systems however, e.g., those that provide some kind of partial synchrony, implementations of such a failure detector do exist [3]. A failure detector acts as a modular black-box of which implementation can change with no impact on the algorithm using it, as long as that implementation ensures the adequate completeness and accuracy properties.

1.2 Motivation: From Crash to Muteness Failures

The motivation of our work is to explore the applicability of the *failure detector* approach to a broader context, where failures are not only crash failures but might have a more malicious nature. In a crash-stop model, it is usually assumed that either a process correctly executes the algorithm that has been assigned to it, or the process crashes and completely stops its execution. Some authors have discussed the applicability of the failure detector approach in system models where processes can crash and recover, and communication links are not always reliable [1, 14, 13, 4]. In each of those models however, the notion of failure (and thus of an incorrect process) is completely independent from the algorithms run by processes in the system. This is no more the case with a Byzantine failure model, i.e., with malicious failures.

A process is said to commit a Byzantine failure when it deviates from the specification of its algorithm [11]: the process might not have necessarily stopped its execution, but might send messages that have nothing to do with those it is supposed to send in the context of its algorithm. Malicious failures are thus intimately related to a given algorithm. To illustrate this point, consider a process q

[1] Hereafter, we sometimes refer to distributed algorithms as protocols.

[2] *Consensus* is a fundamental problem that consists, for a set of processes, in deciding on the same final value among a set of initial values. The FLP impossibility result states that no algorithm can solve consensus in an asynchronous distributed system if one process can fail by crashing [7].

that is part of a set Ω of processes trying to agree on some value. Suppose now that q executes agreement protocol $\mathcal{A}$, which is proven to be correct, whereas all other processes in Ω execute a different agreement protocol $\mathcal{A}$', also proven to be correct. With respect to processes executing $\mathcal{A}$', q is viewed as a Byzantine process, that is a *faulty* process, although it executes correct algorithm $\mathcal{A}$. So, the fact that the notion of failure is relative to a given algorithm in a Byzantine model has an important consequence: it is impossible to achieve a complete separation of the failure detector from the algorithm using it. A consequence is an intrinsic circular dependency: if algorithm $\mathcal{A}$ relies on some failure detector $\mathcal{D}$, the latter must in turn be specified and thus implemented in terms of $\mathcal{A}$ (at least partially).

Our paper can be seen as a first step towards understanding how this circular dependency can be explicitly taken into account, when *specifying* and *implementing* failure detectors in a Byzantine model. On the other side, it is well-know that the consensus problem is at the heart of many other agreement problems. So, having a solution to the consensus yields to solution to other agreement problems like atomic broadcast [3], atomic commitment [9] and membership [8]. Therefore, by allowing to solve consensus, the failure detector allows indirectly to solve other agreement problems that are fundamental in the context of fault-tolerance. Thus, the ultimate motivation behind extending the paradigm of failure detectors to Byzantine model, is to give a correct abstraction in this model, which allows to solve other agreement problems like atomic broadcast.

Muteness Failure Detectors To simplify our discussion, rather than considering all possible Byzantine behaviours, we introduce a weaker malicious failure model that we call the *muteness* failure model. Intuitively, a process p is said to be *mute* with respect to algorithm $\mathcal{A}$ if p stops sending $\mathcal{A}$'s messages to one or more processes. Crash failures are particular cases of mute failures, which are themselves particular cases of Byzantine failures. Interestingly, distinguishing mute failures from other Byzantine failures allows us to clearly separate liveness issues from safety issues: muteness failures are those preventing the progress in consensus algorithms and should be captured at the failure detector level, whereas other kinds of Byzantine failures can be handled at the algorithmic level [12, 5].

We restrict our work to a class of distributed algorithms, which we call *regular round-based algorithms*. Roughly speaking, this class includes all algorithms that have a regular and round-based communication pattern. Most consensus algorithms that make use of unreliable failure detectors belong to that class, including the centralised algorithm of [3] and the decentralised algorithm of [15]. By analogy with crash failure detectors, we define muteness failure detector $\diamond M_{\mathcal{A}}$ as one that tries to capture muteness failures with respect to a given algorithm $\mathcal{A}$. We show that our specification of $\diamond M_{\mathcal{A}}$ does indeed make sense in the context of regular round-based algorithms, and we then describe an implementation of $\diamond M_{\mathcal{A}}$ in a partial synchrony model where there exist bounds on

message latency and clock skew, but these bounds are unknown and hold only after some unknown point in time.

1.3 Contribution: Preserving the Modularity of Failure Detectors

The key contribution of our work is to show that, even though a muteness failure detector is inherently related to a given algorithm, we can still partly preserve the modularity of the failure detector approach. This result is conveyed along four dimensions.

1. From the specification point of view, we define the muteness failure detector $\diamond M_{\mathcal{A}}$ in terms of axiomatic properties. By doing so, we provide an abstraction that helps proving the correctness of consensus algorithms in a Byzantine model.
2. From the algorithmic point of view, the dependency between the failure detector and the algorithm using it can be reduced to a simple well-defined interaction. Modulo this interaction, we can reuse the decentralised consensus algorithm of [15] (initially designed for the crash-stop model) as it is, in a model with mute processes, by merely replacing the failure detector $\diamond\mathcal{S}$ with $\diamond M_{\mathcal{A}}$.
3. From the implementation point of view, we isolate the dependency between some algorithm $\mathcal{A}$ and the implementation of its corresponding $\diamond M_{\mathcal{A}}$ in some function $\Delta_{\mathcal{A}}\ (r)$; the latter is used to increment the timeout at the beginning of each round r. Roughly speaking, the correctness of $\diamond M_{\mathcal{A}}$'s implementation does not only rely on partial synchrony assumptions made on the system, but also on time assumptions made on algorithm $\mathcal{A}$. Intuitively, those assumptions state a set of necessary conditions for finding a function $\Delta_{\mathcal{A}}$ that makes our implementation of $\diamond M_{\mathcal{A}}$ satisfy adequate completeness and accuracy properties.
4. From the practical point of view, defining an adequate abstraction of failure detectors in Byzantine model to solving the consensus problem can be viewed as a cornerstone to solving other agreement problems like atomic broadcast, which is powerful primitive to build fault tolerant applications.

1.4 Roadmap

Section 2 presents our model and formally introduces the notion of muteness failures. Section 3 defines the properties of the muteness failure detector $\diamond M_{\mathcal{A}}$ and specifies the class of algorithms that we consider. Section 4 presents our implementation of $\diamond M_{\mathcal{A}}$. We then prove the correctness of our implementation in a partial synchrony model, provided some timing assumptions on algorithms using $\diamond M_{\mathcal{A}}$. Section 5 shows how a consensus algorithm designed in a crash-stop model, namely the decentralised protocol of [15], can be reused in the context of muteness failures. Finally, Section 6 describes the research results that relate to ours, and Section 7 closes the paper with some concluding remarks and open questions.

2 The Model

This section introduces our model, basically made of processes participating in a distributed algorithm, via the execution of a local automata. We also formally define the muteness failure model that we consider, and discuss how it relates to other failure models.

2.1 Algorithm & Automata

We consider an *asynchronous* distributed system, i.e, there is no upper bound on the time required for a computation or a communication. The system is composed of a finite set $\Omega = \{p_1, ..., p_N\}$ of N processes, fully interconnected through a set of reliable communication channels. Hereafter, we assume the existence of a real-time global clock outside the system: this clock measures time in discrete numbered ticks, which range $\mathcal{T}$ is the set of natural numbers $\mathbb{N}$. We define distributed algorithm $\mathcal{A}$ as a set of deterministic automata $\mathcal{A}_p$ run by processes in the system. We sometimes refer to $\mathcal{A}_p$ as an *algorithm* rather than an automata but it should be clear that we mean the local automata run by some correct process p.

2.2 A Restricted Byzantine Model

Byzantine failures can be split into *undetectable* and *detectable* failures [10]. Undetectable failures are those that cannot be detected from received messages, e.g., a Byzantine process that cheats on its initial value when participating to some agreement protocol. Faced with the impossibility to detect such failure, a process that commits only undetectable failures is considered as correct by the other correct processes. Among detectable failures, we have (1) *commission* failures (with respect to some algorithm), i.e., messages that do not respect the semantics of the algorithm and (2) *omission* failures (with respect to some algorithm), i.e., expected algorithm messages that never arrive. According to this classification, muteness can be seen as a permanent omission failure, i.e., a mute process is a process that stop sending any algorithm messages to one or more correct processes.

Muteness Failures. Each algorithm $\mathcal{A}$ generates a set of messages that have some specific syntax. We say that a message m is an $\mathcal{A}$ message if the syntax of m corresponds to the syntax of the messages that could be generated by $\mathcal{A}$. Note that a message m sent by a Byzantine process that carries a semantical fault but has a correct syntax is considered as an $\mathcal{A}$ message.

Based on this definition, we define a *mute* process as follows. A process q is *mute* with respect to some algorithm $\mathcal{A}$ and some process p if q prematurely stops sending $\mathcal{A}$ messages to p. We say that process q fails by *quitting algorithm $\mathcal{A}$ with respect to some process p*. A muteness failure pattern F is a function from $\Omega \times \mathcal{T}$ to 2^{Ω}, where $F(p,t)$ is the set of processes that quit algorithm $\mathcal{A}$ with respect to p

by time t. By definition, we have $F(p,t) \subseteq F(p,t+1)$. We also define $quit_p(F) = \bigcup_{t \in \mathcal{T}} F(p,t)$ and $correct(F) = \Omega - \bigcup_{p \in \Omega} quit_p(F)$. We say that q is *mute to* p if $q \in quit_p(F)$, and if $q \in correct(F)$, we say q is *correct*.

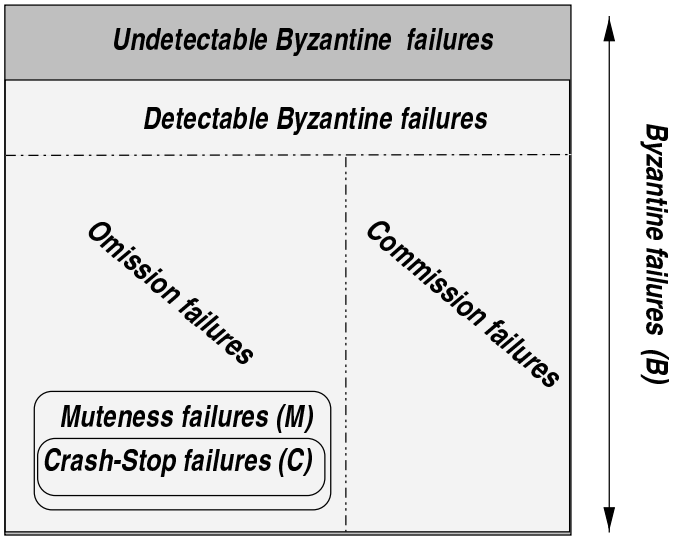

Fig. 1. A Classification of Byzantine Failures

Muteness failures constitute a subset of Byzantine behaviours and a superset of crash-stop failures. Figure 1 conveys this idea, with B denoting the set of all possible Byzantine failures, M the set of muteness failures, and C the set of crash-stop failures. A crashed process is a mute process with respect to all other processes. Note that $M \not\subseteq C$ because a mute process might stop sending messages without crashing.

A Minimum Number of Correct Processes. Let f be the upper bound on the number of faulty processes tolerated in the system. The FLP impossibility shows that in an asynchronous model no agreement problem can be solved if $f > 0$. However, different tacks were proposed in the literature to circumvent this impossibility like extending the asynchronous model with failure detectors, postulating some probabilistic behaviour about the system message, etc. However, all the proposed algorithms assume a minimum number of correct processes, refered hereafter as $N - f$. In the crash-stop model, the value of f is less than $N/2$ and in the Byzantine model this value is less than $N/3$. [3]

[3] For the detailed proof of this result see the work done by Bracha and Toueg in [2].

3 Muteness Failure Detector

This section formally presents the $\diamond M_{\mathcal{A}}$ muteness failure detector and the class of algorithms that can use that detector.

3.1 The $\diamond M_{\mathcal{A}}$ Muteness Failure Detector

A muteness failure detector is a distributed oracle aimed at detecting mute processes. Since the notion of muteness failure is related to some algorithm $\mathcal{A}$, the notion of muteness detector is also related to $\mathcal{A}$. Formally, the $\diamond M_{\mathcal{A}}$ muteness failure detector is expressed in terms of the following two properties:

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

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

3.2 Regular Round-Based Algorithms

There are algorithms for which the use of $\diamond M_{\mathcal{A}}$ makes no sense. More precisely, for such algorithms, it is impossible to implement $\diamond M_{\mathcal{A}}$, even in a completely synchronous model. Intuitively, these algorithms are those for which a mute process cannot be distinguished from a correct process, even in a completely synchronous system, i.e., algorithms where muteness can be a correct behaviour. Therefore, we define here a class of algorithms, named $\mathcal{C}_{\mathcal{A}}$, for which the use of $\diamond M_{\mathcal{A}}$ does indeed make sense. We characterise this class by specifying the set of attributes that should be featured by any algorithm $\mathcal{A} \in \mathcal{C}_{\mathcal{A}}$. We qualify such algorithms as *regular round-based*.

Attribute (a). Each correct process p owns a variable $round_p$ that takes its range $\mathcal{R}$ to be the set of natural numbers $\mathbb{N}$. As soon as $round_p = n$, we say that *process p reaches round n*. Then, until $round_p = n + 1$ process p is said to be *in round n*.

Attribute (b). In each round, there is at most one process q from which all correct processes are waiting for one or more messages. We say that q is *the critical process of round n* and its explicitly awaited messages are said to be *critical messages*.

Attribute (c). With at least $N - f$ correct processes participating in algorithm $\mathcal{A}$, each process p is critical every k rounds, $k \in \mathbb{N}$ and if in addition p is correct then it sends a message to all in that round.

Intuitively, Attribute (a) states that $\mathcal{A}$ proceeds in rounds, while Attribute (b) defines the notion of critical process and restricts the number of such processes to one in each round. Finally, Attributes (c) expresses, in terms of rounds, that a

correct process is critical an infinite number of times, and that it should therefore not be mute. Interestingly, many agreement protocols that we know of feature these three attributes. In particular, both the centralised consensus algorithm of [3] and the decentralised consensus algorithm of [15] are instances of class $\mathcal{C}_{\mathcal{A}}$. They trivially feature Attributes (a), (b) and (c), since they proceed in asynchronous rounds and rely on the rotating coordinator paradigm; in particular, we have $k = N$ for their instance of Attribute (c).

4 An Implementation for $\diamond M_{\mathcal{A}}$

In the crash-stop failure model, the implementation of some failure detector $\mathcal{D}$ can be made independent of the messages sent by the algorithm using $\mathcal{D}$. For example, this is the case of the timeout-based implementation of $\diamond\mathcal{P}$ sketched in [3], which sends periodically its own "p-is-alive" messages: such messages are completely separate from those generated by the consensus algorithm (or by whatever algorithm using the failure detector).[4] This independence is made possible because a crashed process stops sending any kind of messages, no matter whether they are generated by the algorithm or by the local failure detector module. In some sense, we can say that, in a crash-stop model, incorrect processes have a coherent behaviour with respect to the messages they send: if they stop sending messages related to the failure detector, they also stop sending messages related to the algorithm.

In the muteness failure model, on the contrary, incorrect processes can stop sending algorithm messages without crashing. In other words, they can play by the rule as far as messages "p-is-alive" are concerned, and at the same time they can stop sending any other messages. So, the periodic reception of messages from the failure detector module of some process p is no longer a guarantee that an algorithm message will eventually arrive from p. Consequently, an implementation of $\diamond M_{\mathcal{A}}$ based on independent messages cannot help to capture mute processes.

4.1 An Algorithm $\mathcal{I}_{\mathcal{D}}$ for Implementing $\diamond M_{\mathcal{A}}$

Algorithm 1 gives an implementation $\mathcal{I}_{\mathcal{D}}$ of the muteness failure detector $\diamond M_{\mathcal{A}}$. It relies on a timeout mechanism and is composed of three concurrent tasks. Variable Δ_p holds the current timeout and is initialised with some arbitrary value $init_{\Delta} > 0$ that is the same for all correct processes. In addition, $\mathcal{I}_{\mathcal{D}}$ maintains a set $output_p$ of currently suspected processes and a set $critical_p$ containing the processes that p is allowed to add to its $output_p$ set. These two sets are initially empty. A newly suspected process is added to $output_p$ by Task 1 as follows: if p does not receive a "q-is-not-mute" message for Δ_p ticks from some process q that is in $critical_p$, q is suspected to be mute and inserted in the $output_p$ set.

[4] Note that this implementation is correct only if we assume that failure detector messages and algorithm messages do not prevent each other from reaching their destination.

Algorithm 1 Implementation $\mathcal{I}_{\mathcal{D}}$ of Muteness Failure Detector $\diamond M_{\mathcal{A}}$

```
1: {Every process p executes the following :}

2: Δ_p ← init_Δ; output_p ← ∅; critical_p ← ∅;                    {Initialisation}

3: for all q ∈ critical_p do                                        {Task 1}
4:    if (q ∉ output_p) ∧ (p did not receive "q-is-not-mute" during Δ_p ticks) then
5:       output_p ← output_p ∪ q

6: when receive "q-is-not-mute" from A_p                            {Task 2}
7:    if (q ∈ output_p ) then
8:       output_p ← output_p − q

9: when receive new_critical_p from A_p                             {Task 3}
10:   critical_p ← new_critical_p
11:   Δ_p ← g_A(Δ_p)
```

Interactions between $\mathcal{A}_p$ and $\diamond M_{\mathcal{A}}$. Figure 2 sketches how algorithm $\mathcal{A}_p$ executed by a correct process p and $\diamond M_{\mathcal{A}}$ interact: besides queries to $\diamond M_{\mathcal{A}}$ (arrow 1), our implementation $\mathcal{I}_{\mathcal{D}}$ handles two more interactions with $\mathcal{A}_p$. Tasks 2 and 3 are responsible for that. Each time p receives a message from some process q (arrow 2), algorithm $\mathcal{A}_p$ delivers "q-is-not-mute" to $\mathcal{I}_{\mathcal{D}}$ (arrow 3). As a consequence, Task 2 removes process q from $output_p$ in case q was suspected. At the beginning of each round, $\mathcal{A}_p$ delivers a $new\_critical_p$ set to $\mathcal{I}_{\mathcal{D}}$ (arrow 4) containing the *critical processes* of the new round. Task 3 updates $critical_p$ accordingly to the set $new\_critical_p$. In addition, Task 3 also computes a new value for timeout Δ_p by applying some function $g_{\mathcal{A}}$ on the current value of Δ_p. Since the timeout is updated in each new round, there exists a corresponding function $\Delta_{\mathcal{A}} : \mathcal{R} \rightarrow \mathcal{T}$ that maps each round n onto its associated timeout $\Delta_{\mathcal{A}}(n)$. For instance, if function $g_{\mathcal{A}}$ doubles the current timeout Δ_p, then $\Delta_{\mathcal{A}}(n) = 2^{n-1}\ init_{\Delta}$.

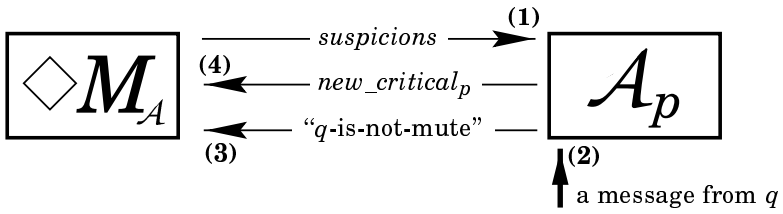

Fig. 2. Interactions between $\mathcal{A}_p$ and $\diamond M_{\mathcal{A}}$

4.2 Assumptions for Proving the Correctness of Algorithm $\mathcal{I}_{\mathcal{D}}$

Our algorithm $\mathcal{I}_{\mathcal{D}}$ does not approximate the bound on communication delays, as does the implementation of $\diamond\mathcal{P}$ in [3], but rather the maximum delay between two consecutive $\mathcal{A}$'s messages sent by some correct process. Therefore, proving the correctness of this implementation, even when assuming a partial synchrony model, is not a straightforward task. Indeed, the delay between two consecutive $\mathcal{A}$'s messages does not depend only on transmission delay, but also depends on the communication pattern of algorithm $\mathcal{A}$.

In order to prove the correctness of implementation $\mathcal{I}_{\mathcal{D}}$, we then rely on (1) partial synchrony assumptions, and (2) time assumptions on algorithm $\mathcal{A}$ and on the timeout function of $\mathcal{I}_{\mathcal{D}}$, i.e., on function $\Delta_{\mathcal{A}}$. In addition, we assume that there are permanently $N - f$ correct processes participating in distributed algorithm $\mathcal{A}$.[5]

Partial Synchrony Assumptions. The *partial synchrony* model assumed here is slightly weaker than those discussed in [6], a reference paper by Dwork et al. on consensus in partial synchrony. In the model they consider, the bound δ on communication delays and the bound ϕ on processes' relative speeds do exist but they are either unknown, *or* known but they hold only after some *Global Stabilisation time*(hereafter GST) that is itself unknown. In the following, we assume a system where δ and ϕ are both unknown *and* hold only after some unknown GST. Such a weaker partially synchronous system is also the one assumed by Chandra and Toueg for their implementability proof of $\diamond\mathcal{P}$ [3]. From now on, we consider the system only after GST, i.e., we assume only values of the global clock that are greater or equal than GST.

Time Assumptions on $\mathcal{A}$ and $\mathcal{I}_{\mathcal{D}}$. In order to state those assumptions, we need a few definitions. We say that some round n is *reached* as soon as at least one correct process p reaches n; once a round is reached, it remains so forever. For some round n we say that n is *completed* if all correct processes have received all the critical messages of round n.

Assumption (a). There exists a constant β such that the following holds. Let n be a round with a correct critical process p. As soon as p is in round n together with at least $N - f$ correct processes, then n is completed in some constant time β.

Assumption (b). There exists a function $h : \mathcal{R} \rightarrow \mathcal{T}$ that maps each reached round n onto the *maximum time* required by any correct process in some round $m < n$ to reach n.

Assumption (c). There exists a function $\Delta_{\mathcal{A}}$ such that the following holds. There exists a round n' such that $\forall n \geq n'$ where n is a reached round, we have:

[5] We will come back to this additional assumption in Section 4.4, where we discuss what happens if it does not hold forever.

$$\Delta_{\mathcal{A}}(n) > h(n) \ \wedge \ \Delta_{\mathcal{A}}(n+1) - h(n+1) > \Delta_{\mathcal{A}}(n) - h(n).$$

Intuitively, this means that after round n', the timeout $\Delta_{\mathcal{A}}(n)$ associated with any reached round $n \geq n'$ is larger and grows faster than $h(n)$.

Corollary 41 *From Assumption (a) to (c), we can easily infer that:*

$$\exists n' \in \mathcal{R}, \forall n \geq n', \ \Delta_{\mathcal{A}}(n) > h(n) + \beta.$$

4.3 Correctness Proof of Algorithm $\mathcal{I}_{\mathcal{D}}$

We now prove that, when used by any regular round-based algorithm, $\mathcal{I}_{\mathcal{D}}$ satisfies the Eventual Mute Completeness and the Eventual Weak Accuracy properties, under the assumptions of Section 4.2. This is formally expressed by Theorem 42.

Theorem 42 *When used by an algorithm $\mathcal{A}$ of class $\mathcal{C}_{\mathcal{A}}$, $\mathcal{I}_{\mathcal{D}}$ ensures the properties of $\diamond M_{\mathcal{A}}$ in the partial synchrony model, under Assumptions (a) to (c).*

Proof: We first prove the Mute Completeness property, which is needed to prove the Eventual Weak Accuracy property.

Eventual Mute Completeness. By Attribute (c) of any algorithm $\mathcal{A} \in \mathcal{C}_{\mathcal{A}}$, we infer that each process is critical in an infinite number of rounds. Therefore, each process q is eventually added the set of $critical_p$ of a correct process p. If q is mute to process p, that means q stops sending messages to p forever. Thus the algorithm $\mathcal{A}_p$ of process p stops receiving messages from q and algorithm $\mathcal{I}_{\mathcal{D}}$ stops receiving "q-is-not mute" messages (Task 2). Therefore, there is a time t after which process p timeouts on q and inserts it in its $output_p$ set (Task 1). Since q stops sending messages to p forever, process q is never removed from $output_p$ and q is suspected forever to be mute by p. Hence, there is a time after which the Eventually Mute Completeness holds forever.

Eventual Weak Accuracy. From the Eventual Mute Completeness a correct process is never blocked forever by a mute process. Therefore, any correct process executes an infinite sequence of rounds. Let p and q be two correct processes. We know that q can only be added to $output_p$ in rounds r where q is the critical process of r. From Corollary 41, we have $\exists n' \in \mathcal{R}, \forall n \geq n', \Delta_{\mathcal{A}}(n) > h(n) + \beta$. Let n be any such round where q is the critical process, and assume that p just reached n, i.e., p is at the beginning of round n. There are two possible cases:

Case 1. Process $q \notin output_p$ at the beginning of round n, i.e, when p starts round n it does not suspect q. Since $\Delta_{\mathcal{A}}(n) > h(n) + \beta$, the timeout Δ_p is larger than the maximum time required by any correct process (includes q) to reach round n, plus the time needed by round n to be completed. As consequence, process p receives the expected critical messages from q without suspecting q. Thus, q is not added to $output_p$. Furthermore, thanks to Corollary 41, we infer that q will not be added to $output_p$ in n, nor in any future round where q will be critical.

Case 2. Process $q \in output_p$ at the beginning of round n, i.e, when p starts round n process q is already suspected by p. Therefore, q was suspected by p in some round $r < n$, where q was r's critical process. Since each correct process executes an infinite sequence of rounds, from the assumption that there are always $N-f$ participating correct processes and from Attribute (c), we know that process q eventually reaches round r as well as at least $N-f$ correct processes and hence sends a message to all in that round. So, there is a round $r' \geq n$, where p eventually receives q's messages and consequently removes q from $output_p$. Since for round r' we also have $\Delta_{\mathcal{A}}(r') > h(r') + \beta$ and $q \notin output_p$, we fall back on Case 1.

Therefore, there exists a round $max(r', n)$ after which process q is never suspected to be mute by any correct process p. Thus, there is a time after which the Eventual Weak Accuracy holds forever. □

4.4 The Problem of Ensuring $\diamond M_{\mathcal{A}}$ Properties Forever

It is worth noting that several distributed algorithms do not ensure the participation of $N-f$ correct processes *forever*. With such an algorithm $\mathcal{A}$, we say that correct process p *terminates* when $\mathcal{A}_p$ yields its last result and p stops participating in $\mathcal{A}$. This is in particular the case for both the centralised and decentralised agreement algorithms described in [3] and [15] respectively. The problem is that, with less than $N-f$ correct processes, we cannot guarantee anymore that a correct process executes an infinite sequence of rounds and sends regular messages to all. As a consequence, $\mathcal{I}_{\mathcal{D}}$ cannot ensure Eventual Weak Accuracy *forever*.

Often, however, such algorithms also guarantee that once a correct process terminates, all correct processes are then able to eventually terminate without using $\diamond M_{\mathcal{A}}$ anymore.[6] In other words, such algorithms only need $\mathcal{I}_{\mathcal{D}}$ to ensure $\diamond M_{\mathcal{A}}$ properties *as long as no correct process terminates*. Again, this is the case of both aforementioned agreement algorithms.

5 Putting $\diamond M_{\mathcal{A}}$ to work

This section gives an example of a distributed agreement algorithm that can be combined with $\diamond M_{\mathcal{A}}$, namely the decentralised consensus algorithm of [15]; this algorithm is also known as *Early Consensus*, hereafter *EC*. We already known that EC is a regular round-based algorithm (Section 3.2). We now show that EC satisfies, in the partial synchrony model defined earlier, Assumptions (a) to (c) (state in Section 4.2). We start by recalling the basic insight of the EC algorithm, and then we proceed with the proofs.

[6] Chandra & Toueg's point out that failure detector properties need only to hold *"long enough for the algorithm to achieve its goal"* [3, page 228].

5.1 Background: Overview of Early Consensus

The detailed EC algorithm can be found in [15], hereafter we outline the basic idea. EC is based on the rotating coordinator paradigm and proceeds in asynchronous rounds, each one being divided into two phases. In Phase 1 of every round r, algorithm EC tries to decide on the estimate of the coordinator p_c of round r. The coordinator p_c starts by broadcasting its current $estimate_c$. When a process receives $estimate_c$, it reissues (broadcasts) this value to all. Once process has received $estimate_c$ from $N - f$ processes, it broadcasts a *decision* message containing $estimate_c$ and decides on it. Phase 1 is illustrated in Figure 3.

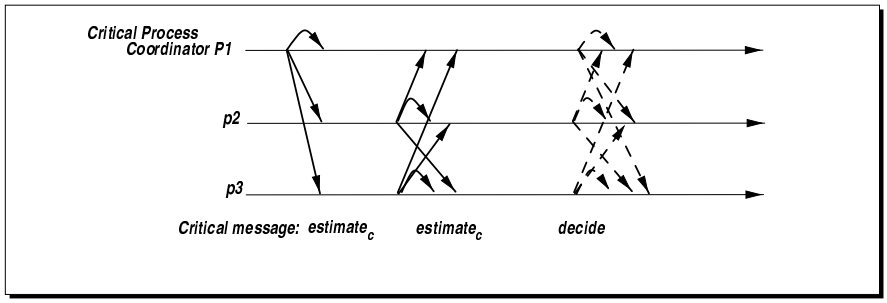

Fig. 3. Phase 1 of Early Consensus

If p_c is suspected by at least $N - f$ processes, Phase 2 ensures that if any process decides on $estimate_c$ in round r, then all correct processes that start round $r + 1$ set their current estimate to $estimate_c$. This is ensured as follows. When a process suspects coordinator p_c, it broadcasts a *suspicion* message. Once a process has received at least $N - f$ *suspicion* messages, it broadcasts its current *estimate* in a so-called *GoPhase2* message. Once a process has received $N - f$ *GoPhase2* messages, it checks if one of the received estimates is the estimate of p_c.[7] If an estimate sent by p_c is found the process adopts it and moves to round $r + 1$. Phase 2 is illustrated in Figure 4.

5.2 Usability Proof of $\diamond M_{\mathcal{A}}$ by Algorithm EC

In this section we show that the EC algorithm satisfies the time assumptions stated in Section 4.2.

Lemma 51 *Assumption (a) holds with $\mathcal{A}$ = EC.*

[7] An estimate is composed of two fields: (1) the value of the estimate and (2) the identifier of the process who proposed this estimate.

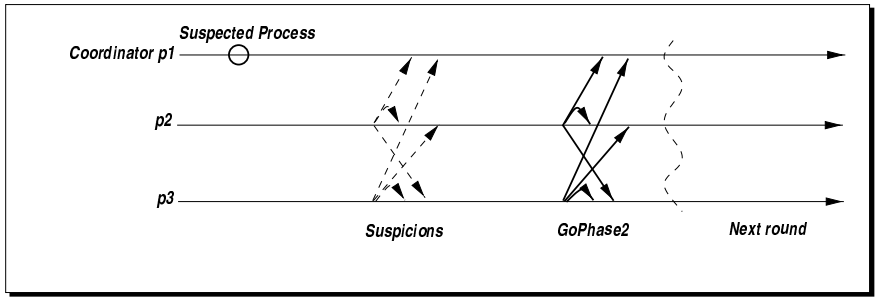

Fig. 4. Phase 2 of Early Consensus

PROOF: Let n be a round with a correct critical process p, and assume that p is is in round n together with at least $N - f$ correct processes. Since we are after GST, process p sends its current estimate in bounded time $\leq \phi$. The estimate is then received by all other correct processes in bounded time $\leq \delta + \phi$. Therefore, round n is completed in a constant time $\beta = \phi + \delta$. □

Lemma 52 *Assumption (b) holds with $\mathcal{A} = EC$.*

PROOF: This proof shows the existence of function h which, as already said, computes the maximum time required by any correct process in some round $m < n$ to reach n. Let p_i be a correct process in the most advanced round, let say n. Let p_j be a correct process in the less advanced round, let say m. In EC, each correct process sends a *GoPhase2* message to all before proceeding to the next round. Therefore, process p_i already sent message $(p_i, m, 2, -)$ to all, when it was in round m.

Since we are after GST, message $(p_i, m, 2, -)$ is received by each correct process in a bounded time $\leq \delta$. Each correct process delivers this message and relays it to all the other processes in bounded time $\leq (N+1)\phi$. So, any correct process, in particular process p_j which is in round m, collects $N - f$ *GoPhase2* messages and proceeds to the next round in bounded time $\leq (2\delta + (N + 1)\phi)$. Therefore, process p_j reaches round n in bounded time $\leq (n - m)(2\delta + (N + 1)\phi)$. Since the worse case is $m = 1$, we can roughly bound the time needed by process p_j to reach round n by $(n - 1)(2\delta + (N + 1)\phi)$. We have thus proved the existence of a linear function $h(n) = (n - 1)(2\delta + (N + 1)\phi)$ for algorithm EC. □

Lemma 53 *Assumption (c) holds with $\mathcal{A} = EC$.*

PROOF: Immediate from Lemma 52, if we define exponential function $\Delta_{\mathcal{A}}(n) = 2^{n-1} init_{\Delta}$. □

Theorem 54 *If $\mathcal{A} = EC$ and $\Delta_{\mathcal{A}}(n) = 2^{n-1} init_{\Delta}$, $\mathcal{I}_{\mathcal{D}}$ ensures the properties of $\diamond M_{\mathcal{A}}$ in the partial synchrony model.*

PROOF: Immediate from Lemmas 51 to 53, and from Theorem 42 □

6 Related Work

Beside our work, we know of two research efforts that aim at extending the notion of failure detectors to Byzantine models and solving consensus using such detectors. Malkhi and Reiter considered a system model where all messages are exchanged using a causal-order reliable broadcast primitive [12]. They defined the notion of *quiet process* that is close to our notion of *mute process*. They also introduced a failure detector, noted $\diamond\mathcal{S}$ (bz), and they expressed its properties in terms of *Strong Completeness* and *Eventual Weak Accuracy*. The aim of $\diamond\mathcal{S}$ (bz) is to track processes that prevent the progress of the algorithm using it. The failure detector $\diamond\mathcal{S}$ (bz) is shown to be strong enough to solve consensus.

In [10], Kihlstrom et al. define two classes of failure detectors: (1) the eventual Weak Byzantine failure detector $\diamond\mathcal{W}$ (Byz), and (2) the eventual Strong Byzantine failure detector $\diamond\mathcal{S}$ (Byz). Both detectors are shown to be strong enough to solve consensus. Contrary to the specification proposed by Malkhi and Reiter, and contrary to the one we propose, these classes of failure detectors capture all the *detectable* Byzantine failures. Consequently among the set of failures captured by $\diamond\mathcal{W}$ (Byz) or $\diamond\mathcal{S}$ (Byz) there are failures that prevent the progress of the algorithm, i.e., muteness failures (in point-to-point communication network) or quietness failures (in causal-order reliable broadcast communication network).

Neither [12] nor [10] address the fundamental circularity problem (i.e., the dependency between the algorithm and the Byzantine fault detector), which we believe is a fundamental issue when applying the failure detector approach in the context of Byzantine failures. This is probably because none of those papers discuss the correctness of Byzantine fault detector implementations. In [12], the authors did not address the implementability issues, while authors of [10] present a non-proved implementation that roughly aims to insert timeout mechanism in the branch of the algorithm where there are expected messages: but this is far from proving that this implementation provides the expected properties of the failure detector. By focusing on mute detectors and restricting our work to regular round-based algorithms, we could address the circularity problem and describe a correct implementation of mute detectors in a partial synchrony model.

7 Concluding Remarks

The motivation of this paper was to extend failures detectors for crash-stop failures to malicious failures. The extension was however not straightforward because, in a malicious environment, the notion of fault is intimately related to a given algorithm. It is thus impossible to specify a Byzantine fault detector that is independent from the algorithm using it (unlike in a crash-stop model). Furthermore, given the necessity of a two-ways interaction between the fault detector and the algorithm, one might end up with a fault detector that is impossible to implement, even in a completely synchronous model. This paper can be viewed as a first step towards better understanding these issues in a Byzantine

environment. The paper focuses on muteness failures. Muteness failures are malicious failures in which a process stops sending algorithm messages, but might continue to send other messages. The paper presents both a definition of a mute detector $\diamond M_{\mathcal{A}}$ and a protocol for implementing $\diamond M_{\mathcal{A}}$ in a partial synchrony model. The mute detector $\diamond M_{\mathcal{A}}$ is strong enough to solve consensus in an asynchronous distributed system with mute failures. Although the implementation of $\diamond M_{\mathcal{A}}$ and the algorithm using it must cooperate, which is actually inherent in the problem, we have confined that interaction inside a specific module. Furthermore, we have shown that, modulo that interaction, one can reuse a consensus algorithm designed in a crash stop model.

Open Questions We restricted our work to the class of *regular round-based* distributed algorithms. Although many algorithms belong to that class, it would be interesting to see whether one could extend this class and still be able to provide a sensible implementation of $\diamond M_{\mathcal{A}}$ in a partial synchrony model. In fact, some of the attributes of that class could indeed be relaxed. For instance, we required (*) *the permanent participation of a majority of correct processes in the algorithm.* This assumption is not really needed as long as, once a correct process terminates, all correct processes are able to eventually terminate without using $\diamond M_{\mathcal{A}}$ anymore. In other words, $\diamond M_{\mathcal{A}}$ properties are actually only needed *as long as no correct process terminates.* We can thus replace (*) by (**) *there is at least a majority of correct processes participating in the algorithm, as long as no correct process terminates.* It would be interesting to find out whether other attributes can be relaxed and whether the time assumptions that we require to implement $\diamond M_{\mathcal{A}}$ are just *sufficient* or *necessary.*

References

1. M. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. In *12th International Symposium on Distributed Computing.* Springer Verlag, LNCS 1499, September 1998.
2. G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *Journal of the Association for Computing Machinery*, 32(4):824–840, October 1985.
3. T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.
4. Danny Dolev, Roy Friedman, Idit Keidar, and Dahlia Malkhi. Failure detectors in omission failure environments. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing*, page 286, Santa Barbara, California, August 1997.
5. A. Doudou and S. Schiper. Muteness detectors for consensus with byzantine processes (brief announcement). In *Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing (PODC'98)*, Puerto Vallarta, Mexico, June 1998. ACM. An extended version of this brief annoucement is available as a Technical Report, TR 97/230, EPFL, Detp d'Informatique, October 1997, under the title "Muteness Failure Detector for Consensus with Byzantine Processes".
6. C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, apr 1988.

7. M. Fischer, N. Lynch, and M. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32:374–382, April 1985.
8. R. Guerraoui and A. Schiper. Consensus service: a modular approach for building agreement protocols in distributed systems. In *IEEE 26th Int Symp on Fault-Tolerant Computing (FTCS-26)*, pages 168–177, June 1996.
9. Rachid Guerraoui. Revisiting the relationship between non-blocking atomic commitment and consensus. In Jean-Michel Hélary and Michel Raynal, editors, *Distributed Algorithms, 9th International Workshop, WDAG '95*, volume 972 of *Lecture Notes in Computer Science*, pages 87–100, Le Mont-Saint-Michel, France, 13–15 September 1995. Springer.
10. Kim Potter 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.
11. L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.
12. D. Malkhi and M. Reiter. Unreliable Intrusion Detection in Distributed Computations. In *Proc. 10th Computer Security Foundations Workshop (CSFW97)*, pages 116–124, June 1997.
13. O.Babaoğlu, R.Davoli, and A.Montresor. Failure Detectors, Group Membership and View-Synchronous Communication in Partitionable Asynchronous Systems. Technical Report UBLCS-95-18, Department of Computer Science University of Bologna, November 1995.
14. R. Oliveira, R. Guerraoui, and A. Schiper. Consensus in the crash-recover model. Technical Report 97/239, École Polytechnique Fédérale de Lausanne, Switzerland, August 1997.
15. A. Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3):149–157, April 1997.