Encapsulating Failure Detection: from Crash to Byzantine Failures

Assia Doudou⋆, Benoît Garbinato⋆⋆, and Rachid Guerraoui

Distributed Programming Laboratory,
Swiss Federal Institute of Technology, Lausanne.

Abstract. Separating different aspects of a program, and encapsulating them inside well defined modules, is considered a good engineering discipline. This discipline is particularly desirable in the development of distributed agreement algorithms which are known to be difficult and error prone. For such algorithms, one aspect that is important to encapsulate is failure detection. In fact, a complete encapsulation was proven to be feasible in the context of distributed systems with process crash failures, by using *black-box* failure detectors. This paper discusses the feasibility of a similar encapsulation in the context of *Byzantine* (also called *arbitrary* or *malicious*) failures. We argue that, in the Byzantine context, it is just impossible to achieve the level of encapsulation of the original crash failure detector model. However, we also argue that there is some room for an intermediate approach where algorithms that solve agreement problems, such as consensus and atomic broadcast, can still benefit from *grey-box* failure detectors that partially encapsulate Byzantine failure detection.

1 Introduction

Agreement problems are at the heart of reliable distributed computing, in particular transactional and real-time distributed systems. A canonical form of such problems is the well-known *consensus* problem that has been extensively studied in the literature (See [5] for a survey). Designing agreement algorithms and proving them correct is particularly difficult because of the possibility of partial failures. Roughly speaking, to reach agreement despite failures, processes need some plausible hints about which process is faulty and which process is correct. In a completely asynchronous system, where process relative speeds and message transmission delays are unbounded, obtaining any meaningful hint is impossible. One might never be able to distinguish a process that is faulty from one that is very slow. Consequently, even under the assumption that at most one process may crash, and even in its simplest form, consensus cannot be solved [4]. This fundamental result is known in the literature as the *FLP* impossibility result.

Some timing assumptions about the system are needed in order to solve agreement. The traditional approach consisted in building agreement algorithms that directly manipulate timing parameters to detect process failures, e.g., using timeouts and clock synchronization protocols [3]. Direct handling of timing assumptions makes the resulting (agreement) algorithms very hard to prove and error prone. Adding a simple instruction can for instance cancel the validity of a complicated proof.

A more modular approach was proposed recently [1]. The idea is to encapsulate the timing assumptions inside specific modules called *failure detectors* and specified through precise axiomatic properties. With this approach, an agreement algorithm does not need to directly handle timing assumptions. It just needs to interact with a failure detector through a simple and well defined interface. Obeying the tradition of good practices in software engineering, once a failure detector is specified and implemented, the algorithms that use it only need to know its interface. As long as the properties of the failure detector are ensured, one can freely change its implementation and, for instance, optimize it according to the network topology.

⋆ Currently with Kudelski, Lausanne, Switzerland.
⋆⋆ Currently with Sun Microsystems, Gland, Switzerland.

However, so far, the failure detector approach of [1] was only shown to be feasible in the context of *crash* failures: the simplest form of failures. A process that crashes is simply supposed to halt all its activities. In practice, processes might exhibit more complex behaviors. They may stop only *some*, but not all of their activities, e.g., within the same process, some threads are indefinitely blocked while others are not. Because of software bugs or malicious behaviors, processes may even send messages that have nothing to do with those they are supposed to send. In short, processes can exhibit so-called *Byzantine* failures [10] (sometimes called *arbitrary* or *malicious* failures). Solving agreement in this context is harder than in the crash-failure model, and one obviously needs timing assumptions here as well.

A question is now in order: does the modular failure detector approach of [1] apply to the context of Byzantine failures? To address this question, we proceed in four steps. In each step, we consider one of the fundamental modularity aspects of the original crash failure detector approach of [1], and we discuss the extent to which this aspect remains valid in the context of hypothetical Byzantine failure detectors. The aspects we consider are the following.

– *Specification orthogonality.* The specification of crash failure detectors is independent from the algorithms using such failure detectors. In the Byzantine context, no failure detector specification can feature this aspect. We show that this is due to the inherent nature of Byzantine failures. We propose however an intermediate approach where only part of the Byzantine behaviors is captured by the failure detector, namely *muteness* failures. This approach enables us to capture a tricky part of Byzantine failures inside well defined modules (muteness failure detectors), hence restricting the dependency between the algorithm and the failure detector.

– *Implementation generality.* Due to the specification orthogonality of crash failure detectors, their implementation can be made independent from the algorithms using them. This makes it possible to view a failure detector as a black-box of which implementation can change without any impact on the algorithms that use it, as long as it complies with its specification, i.e., it ensures its axiomatic properties. Since Byzantine failure detectors cannot be specified orthogonally to the algorithms using them, it is not surprising that their implementation depends somehow on those algorithms. We show that this dependency even restricts the set of algorithms that can make a meaningful use of failure detectors. We suggest a *grey-box* approach with a parameterized implementation of a muteness failure detector that is sufficient to solve consensus in a Byzantine environment.

– *Failure Encapsulation.* In [1], all aspects related to failure detection are encapsulated inside the failure detector module. Consequently, the only places where the algorithm deals with failures is by interacting with the failure detector. In a Byzantine context, failures have many faces, some of which can simply not be encapsulated inside the failure detector. We dissect the tricky issues underlying this problem and we present a consensus algorithm that relies on some additional modules to handle those issues. Besides muteness failure detectors, we make use for instance of echo broadcast and certification protocols.

– *Failure Transparency.* Because of the simple nature of crash failures, one can mask them. For example, using a consensus algorithm as an underlying black-box, one can build on top an atomic broadcast algorithm that simply ignores failures, i.e., it does not even need to interact with the failure detector module (the latter is used as sub-module by the consensus algorithm). We argue that this transparency aspect is simply impossible in a Byzantine context. We present however an atomic broadcast algorithm where some failures are masked by the consensus algorithm, whereas others are handled directly.

By discussing these four aspects, this paper contributes to better understanding the applicability of a failure detector approach in a Byzantine environment. We discuss a new failure detection notion and we introduce two Byzantine-resilient agreement algorithms to illustrate our point: a consensus and an atomic broadcast. To simplify the presentation of the paper, the correctness proofs of the algorithms are left to the appendices.

2 Specification Orthogonality

The specification of crash failure detectors is independent from the algorithms using these failure detectors. We call this nice aspect *specification orthogonality*. Achieving the same level of specification orthogonality in a Byzantine environment is not possible, but there is room for an intermediate approach based on a notion of *muteness failure detectors*.

2.1 Failure Detector Specification in a Crash Model

Intuitively, a crash failure is a physical event that causes a process to halt all its activities. A failure pattern is represented by a function that describes crashes that occur in the system, and the failure detector is a distributed oracle that provides hints about the failure pattern [1]. A process p typically consults the failure detector to know whether a given process q has crashed or not. Several classes of failure detectors were defined in [1], each one characterized by a *completeness* and an *accuracy* property. Completeness expresses the ability of a failure detector to eventually detect failures, while accuracy expresses its ability to eventually avoid false suspicions. For instance, the class $\diamond\mathcal{S}$ (*Eventually Strong*) gathers failure detectors which satisfy two fundamental properties: (1) (Strong Completeness) eventually, every process that crashes is permanently suspected by every correct process; and (2) (Eventual Weak Accuracy) eventually, some correct process is never suspected by any process. These axiomatic properties capture the timing assumptions that are sufficient to solve the consensus problem [**?**].

Defined this way, crash failures are *context-free*, i.e., they are defined independently of what is executed by the processes on which they occur, e.g., a consensus algorithm. So, crash failure detectors are specified in a completely orthogonal manner with respect to the algorithms using them.

2.2 Two Flawed Approaches

A Naive Approach. It is very tempting to define Byzantine failure detectors in an analogous way to crash failure detectors. This would mean however defining Byzantine failure events in an analogous way to crash failure events. However, when a process commits a crash event, it stops *all* its activities with respect to *all* other processes in the system. This is clearly not the case in the Byzantine failure model. In such a model, a faulty process might not have necessarily halted, but might for instance send messages that have nothing to do with those it is supposed to send in the context of its algorithm, or it might arbitrarily decide to stop sending any messages (without halting though). In fact, a Byzantine process is a process that arbitrarily decides to deviate from the specification of its algorithm. To illustrate this point, consider a process q that is part of a set Ω of processes trying to agree on some value. Suppose now that q executes agreement algorithm $\mathcal{A}$, which is proven to be correct, whereas all other processes in Ω execute a different agreement algorithm $\mathcal{A}'$, 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 $\mathcal{A}$). In fact, a Byzantine process q might exhibit an incorrect behavior with respect to some process p but behaves correctly with respect to another process p'. For instance, q might be executing $\mathcal{A}$ against process p and $\mathcal{A}'$ against process p.

The specification of Byzantine failures is hence intimately related to a specific algorithm and to a specific process. Therefore, Byzantine failures are not context-free, unlike crash failures. Consequently, a Byzantine failure detector specification cannot be, unlike a crash failure one, orthogonal to the algorithm using it. In short, trying to specify a failure detector that tracks all Byzantine failures introduces too much dependency with the algorithm aimed at using that failure detector.

A Conservative Approach. A different approach consists in separating crash failures from other kinds of Byzantine failures, and in using crash failure detectors to detect the former while leaving the latter to the algorithm itself. The rationale behind this approach is to keep intact the modularity of the original model of [1] and deal with failures that are of a more malicious nature at a separate level, e.g., within the agreement algorithm itself.

Unfortunately, this conservative approach does not make sense either, and this is because crash failure detectors are inherently defined to track physical crashes and not logical failures. To better understand this point, assume that an algorithm designer is provided with a perfect crash failure detector in a Byzantine environment [1]. Such a failure detector guarantees (1) (Strong Completeness) eventually, every process that crashes is permanently suspected by every correct process; and (2) (Strong Accuracy) no process is suspected before it crashes.

Such a failure detector guarantees that every process that crashes is eventually suspected by all processes, and that no process is suspected unless it crashed. The malicious thing about Byzantine processes however is that they can stop participating in the algorithm without crashing. From a more practical point of view, this means that these processes can for instance send *heartbeat* messages in a timely manner (in order not to be suspected to have crashed), yet do not send any message related to the actual algorithm. According to the definition of a perfect failure detector, these are correct processes and there is no reason to suspect them. In some sense, such Byzantine processes do not *physically* crash but they *algorithmically* crash. From the perspective of a correct process however, either behavior can lead to the same problem if the faulty process is not properly detected: the progress of correct processes cannot be guaranteed and even a perfect crash failure detector is useless here.

2.3 An Intermediate Approach: Muteness Failure Detectors

As an alternative to the two flawed approaches sketched above, we suggest an intermediate one that focuses on the notion of *muteness* failures. These can be viewed as special case of Byzantine failures, yet are more general than crash failures (see Figure 1). Such failures encompass what we above referred to as physical *and* algorithmic crashes. Interestingly, adequately defined, muteness failure detectors encapsulate a sufficient level of synchrony to solve the consensus problem in the Byzantine failure model, as we discuss in Section 4. The specification of muteness failure detectors is not orthogonal to the algorithms using them, but it introduces a well-defined dependency relation with the algorithms. Below, we formally define the notions of *muteness failure* pattern and *muteness failure detector*.

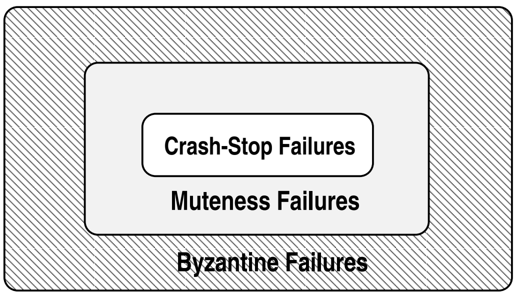

Fig. 1. Inclusion relation between different types of failures

Formal Definitions. Each algorithm $\mathcal{A}$ can be made to generate a set of messages that have some specific syntax, i.e., a syntax that is specific to $\mathcal{A}$. We say that a message m is an $\mathcal{A}$ message if the syntax of m corresponds to the syntax of the messages that may 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 to be 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 process p*. A muteness failure pattern $F_{\mathcal{A}}$ is a function from $\Omega \times \mathcal{T}$ to 2^{Ω}, where $F_{\mathcal{A}}(p,t)$ is the set of processes that quit algorithm $\mathcal{A}$ with respect to p by time t. By definition, we have $F_{\mathcal{A}}(p,t) \subseteq F_{\mathcal{A}}(p,t+1)$. We also define $quit_p(F_{\mathcal{A}}) = \bigcup_{t \in \mathcal{T}} F_{\mathcal{A}}(p,t)$ and $correct(F_{\mathcal{A}}) = \Omega - \bigcup_{p \in \Omega} quit_p(F_{\mathcal{A}})$. We say that q is *mute to* p for $\mathcal{A}$ if $q \in quit_p(F_{\mathcal{A}})$, and if $q \in correct(F_{\mathcal{A}})$, we say q is *correct* for $\mathcal{A}$. We define a *muteness failure detector history* as a function $H_{\mathcal{A}}$ from $\Omega \times \mathcal{T}$ to 2^{Ω} where $H_{\mathcal{A}}(p,t)$ denotes the set of processes suspected by process p to be mute at time t. A *muteness failure detector* is then a function $\mathcal{D}_{\mathcal{A}}$ that maps each failure pattern $F_{\mathcal{A}}$ to a set of failure detectors histories.

We define a class of muteness failure detectors, denoted by $\diamond M_{\mathcal{A}}$, by introducing the *mute $\mathcal{A}$-completeness* and *eventual weak $\mathcal{A}$-accuracy* properties, which should be satisfied by any failure detector $\mathcal{D}_{\mathcal{A}} \in \diamond M_{\mathcal{A}}$. These properties are stated as follows:

Mute $\mathcal{A}$-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 $\mathcal{A}$-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 Implementation Generality

Intuitively, the implementation of a crash failure detector encapsulates the (partial) synchrony of the system [1]. Given some timing assumptions about the underlying system, one can implement a crash failure detector. Once implemented, the latter becomes a very *general* black-box that can be used by *any* algorithm. We call this nice property *implementation generality*. We argue below that this property cannot be achieved in the context of Byzantine failures. One needs to restrict the set of algorithms that can use a given failure detector. We discuss below this restriction and we sketch a parameterized implementation of a muteness failure detector in a partial synchrony model.

3.1 Algorithmic Restrictions

As we pointed out, the specification of a Byzantine (or a muteness) failure detector is context sensitive: it depends on the algorithm using that failure detector. Not surprisingly, the implementation of any failure detector of class $\diamond M_{\mathcal{A}}$ is tightly related to the algorithm $\mathcal{A}$ that uses it. A natural question is now in order: can we devise an implementation of a muteness (or Byzantine) failure detector where the dependency with the algorithm using the failure detector would be captured by a generic parameter? This would mean that *any* algorithm could use such an implementation, provided an adequate instantiation of that generic parameter.

The answer is *no*, as we discuss below in the context of $\diamond M_{\mathcal{A}}$. Basically, there are algorithms for which no implementation of $\diamond M_{\mathcal{A}}$ does make sense. These are algorithms $\mathcal{A}$ in which the expected correct behavior of a process can be confused with an incorrect one, i.e., where the set of correct processes overlaps the set of mute processes. Imagine for example that a correct process p, to respect the specification of $\mathcal{A}$, must become mute with respect to all processes. Then *Mute A-Completeness* requires the processes to suspect p while *Eventual Weak A-Accuracy* requires the processes to eventually stop suspecting p. So, the two properties of the muteness failure detectors become in contradiction. To circumvent this problem, we impose some restrictions on algorithms that can use failure detectors of class $\diamond M_{\mathcal{A}}$. Interestingly, such restrictions do not exclude useful algorithms that solve hard problems like consensus (next section).

Roughly speaking, we require from such algorithms that they have a regular communication pattern, i.e., each correct process communicates regularly (to prevent the case where muteness is viewed as a correct behavior). More precisely, we define a class of algorithms, named $\mathcal{C}_{\mathcal{A}}$, for which the use of $\diamond M_{\mathcal{A}}$ does indeed make sense. We characterize $\mathcal{C}_{\mathcal{A}}$ 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* and we define the attributes that characterize these algorithms as follows.

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 *a critical process of round n* and its explicitly awaited messages are said to be *critical messages.*

Attribute (c). Each process p is critical every k rounds, $k \in \mathbb{N}$ and if p is correct then p 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, the very fact that a correct process is critical an infinite number of times, and that it should therefore not be mute. It is important to notice here that the period of interest of this attribute is the duration of an algorithm, i.e., infinity actually means in pract ice until the algorithm using the failure detector terminates. Interestingly, many agreement algorithms that we know of feature these three attributes. In particular, the consensus algorithm of [1] using $\diamond\mathcal{S}$ is of class $\mathcal{C}_{\mathcal{A}}$ (as well as all variants of that algorithm that were devised later). They trivially feature Attributes (a), (b) and (c), since they proceed in asynchronous rounds and rely on the rotating coordinator paradigm; in particular, they feature $k = N$ for Attribute (c).

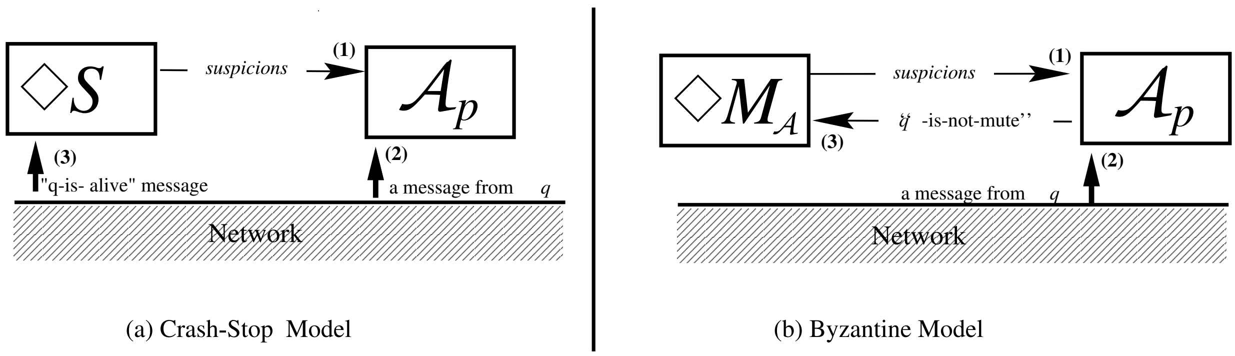

Fig. 2. Algorithm and failure detector modules interaction

3.2 Failure Detector Implementation in a Partial Synchrony Model

To implement a crash failure detector that solves an agreement problem like consensus, one needs to make some timing assumptions about the underlying system model. [1]

In the partial synchrony model of [3] for example, one can provide a timeout-based implementation of $\diamond\mathcal{S}$ [1] (see Figure 2.a). Roughly speaking, this model assumes that δ, the bound on communication delays, and ϕ, the bound on process relative speeds, do exist but that they are either unknown, *or* known but they hold only after some *Global Stabilization time* (GST) that is itself unknown. The implementation of $\diamond\mathcal{S}$ (in [1]) is based on timeouts and proceeds in the following way. Every process p periodically sends "p-is-alive" messages: these *heartbeat* messages

[1] A crash failure detector that would solve consensus, and yet be implementable in an asynchronous model, would obviously contradict the FLP impossibility result [4].

are completely separated from those generated by the consensus algorithm (or by whatever algorithm using the failure detector). This independence is made possible because a crashed process stops sending all kinds of messages, no matter whether they are generated by the algorithm or by the local failure detector module. Therefore, from the non-reception of "p-is-alive" messages, any crashed processes is eventually suspected by all correct processes. The existence of GST plus the periodic sending of "p-is-alive" messages allow to eventually stop suspecting any correct process. So, this implementation meets the required accuracy and completeness properties of $\diamond\mathcal{S}$.

Consider the case of a mute process that stops sending algorithm messages without crashing, but keeps sending other messages, in particular those related to the failure detector implementation. The process plays by the rule as far as messages "p-is-alive" are concerned, and at the same time, stops sending all other messages. So, the periodic reception of messages from the remote 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 capture mute processes.

We still assume the same partial synchrony model as in [1], and we suggest an approach that captures the interaction between the muteness detector and the algorithm within a generic parameter. We consider *regular round-based* algorithms and we give an implementation of $\diamond M_{\mathcal{A}}$ in this context.

The basic idea of our $\diamond M_{\mathcal{A}}$ implementation is sketched in Figure 2.b. Each correct process p holds a timeout Δ_p initialized with some arbitrary value being the same for all correct processes. The value of Δ_p is increased at the beginning of each new round of algorithm $\mathcal{A}_p$ by some function g_A that depends on $\mathcal{A}$ (this is the parameter of our implementation). Besides Δ_p, each failure detector module $\diamond M_{\mathcal{A}}$ maintains a set $output_p$ of currently suspected processes. Each time an $\mathcal{A}$ message is received from some process q (arrow 2), algorithm $\mathcal{A}_p$ delivers *"q-is-not-mute"* to $\diamond M_{\mathcal{A}}$ (arrow 3) and q is removed from $output_p$ in case it was previously suspected. A newly suspected process q is added to $output_p$ (arrow 1) if p does not receive a *"q-is-not-mute"* message for Δ_p ticks (arrow 3).

Proving the correctness of this implementation is not straightforward because, rather than approximating the delay between two *"p-is-alive"* messages (as done for $\diamond\mathcal{S}$ in [1]), our implementation approximates the delay between two $\mathcal{A}$ messages. This task is particularly difficult because the delay between two $\mathcal{A}$ messages could be function of the timeout. For example, each time the timeout increases, the delay between two $\mathcal{A}$ messages could also increase. This would lead to the impossibility of approximating this delay. In [**?**], we prove the correctness of our implementation by assuming (1) a partially synchronous model, and (2) timing properties on $\mathcal{A}$ that hold after GST. Those timing properties state a set of necessary conditions on the timeout function, i.e., g_A , and the existence of a function expressing the progress of any correct process.

In short, we have argued in this section that the implementation of Byzantine (or muteness) failure detectors (like $\diamond M_{\mathcal{A}}$) does not make sense for any arbitrary algorithm $\mathcal{A}$. We suggested some realistic restrictions on the algorithms that would make such implementation possible. Even given those restrictions, we have pointed out the very fact that it is not enough to make timing assumptions about the system model to implement a Byzantine failure detector. One needs to make timing assumptions about $\mathcal{A}$ as well.

4 Failure Encapsulation

Crash failure detectors encapsulate all aspects related to failures. Consequently, a failure detector module acts as a black-box of which only the interface needs to be known by the algorithms that use it. We call this nice aspect *failure encapsulation.* Achieving the same level of failure encapsulation in a Byzantine environment does not seem to be possible, but there is room for an intermediate approach as we discuss in this section. We illustrate our point in the context of the consensus problem: each process proposes a value and the correct processes need to decide the same value among the set of proposed values.

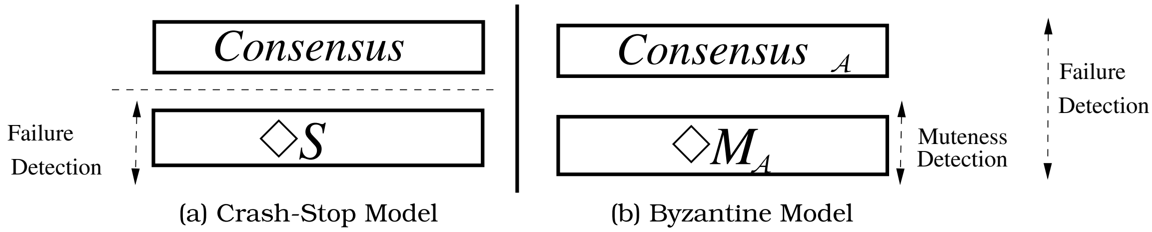

Fig. 3. Failure detection encapsulation

4.1 Failure Encapsulation in the Crash Model

In the consensus algorithm of [1] using $\diamond\mathcal{S}$, all issues related to failures are encapsulated within the failure detector (see Figure 3.a). The algorithm is based on the rotating coordinator paradigm. It proceeds in asynchronous rounds, in a sequential manner (from the perspective of any given process). Each round has exactly one coordinator process. When they reach a given round, the processes wait for the coordinator of that round to propose a value. They wait until they suspect the coordinator of that round to have crashed. If they receive the proposal of the coordinator, they adopt that proposal and acknowledge that adoption. The coordinator decides its value if a majority of processes have adopted that value, and in that case broadcast the decision message to all. In the case where a process in a given round suspects the coordinator of that round, the process moves to the next round.

The critical point in such algorithm is when all processes wait for an estimate from the coordinator. A possible crash of the coordinator might prevent the correct processes from progressing. Each correct process hence consults the output of its failure detector to get hints about possibly crashed processes. From the perspective of the consensus algorithm of [1], the failure detection mechanism is completely hidden within the failure detector module, i.e., within $\diamond\mathcal{S}$.

4.2 Failure Encapsulation in the Byzantine Model

When considering a Byzantine model, obviously a complete encapsulation of failures cannot be provided. We illustrate this point in the context of consensus in a Byzantine model augmented with $\diamond M_{\mathcal{A}}$ instead of $\diamond\mathcal{S}$. Indeed, $\diamond M_{\mathcal{A}}$ can only cope with the muteness failures, while leaving other Byzantine behaviors undetected (see Figure 3.b). More precisely, the other failures are the following:

– *Conflicting Messages.* The consensus algorithm of [1] is based on the rotating coordinator paradigm, i.e., in each round, the current coordinator communicates its estimate to all processes. Considering this algorithm in the context of Byzantine failures raises a new problem: a Byzantine coordinator, rather than sending a unique estimate to all processes, may send them different estimates. These messages are what we refer to as *conflicting* messages. The problem here is to prevent correct processes from delivering, in the same round, conflicting messages from a Byzantine coordinator. This is an instance of the well-known *Byzantine Generals* problem [10].

– *Invalid Messages.* In order to define the notion of invalid messages, we first introduce the notion of an event *preceding* another event. Let $e1$ be the event of receiving a set of messages $m1$ at 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 $e1 \prec e2$, if the event $e2$ of sending $m2$ is conditioned by the event $e1$ of receiving set $m1$ (see Figure 4). From an algorithmic viewpoint, this definition can be translated as follow: If receive $m1$ then send $m2$. In a trusted model, i.e., one that excludes malicious behaviors, when some process performs some event $e2$, such that $e1 \prec e2$,

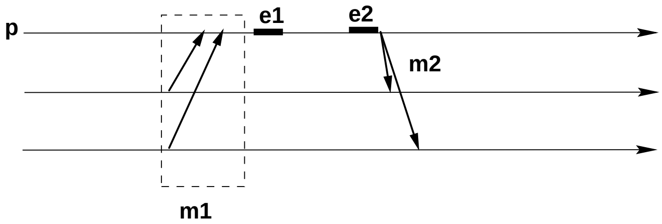

Fig. 4. $e1$ precedes $e2$

it is trivially ensured that (1) $e1$ happened before $e2$, and (2) if the computation of message $m2$ depends on set $m1$ then $m1$ is correctly taken into account to compute $m2$. In contrast, this is no longer true 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 the Byzantine process is referred to as an invalid message.

– *Missing Messages.* A *Missing* message is a message that was voluntary skipped by some incorrect process, while this process is still sending other messages.

4.3 Solving Consensus in the Byzantine Model

In the absence of a way to completely encapsulate the detection of Byzantine failures into a single module, we propose an intermediate approach that aims at dealing with the malicious failures stated above via a modular strategy.

Our algorithm is obtained by transformation of the crash-resilient consensus algorithm of [1]. The main adaptations can be summarized as follows: (1) replacing $\diamond\mathcal{S}$ by $\diamond M_{\mathcal{A}}$ to cope with muteness failures, (2) applying an *echo broadcast* mechanism in order to cope with conflicting messages, (3) appending a *certificate* to some messages in order to cope with invalid messages, and (4) detecting skipped messages. We also assume that f, the maximum number of Byzantine processes in a run, is less than two thirds of the total number of processes N, i.e., the processes involved in consensus. Since the need of adaptation (1) has been extensively discussed in Section 2, we restrict our discussion to adaptations (2) to (4). Note also that, in our adaptation, we found it convenient to consider a decentralized variant of the original consensus algorithm of [1]: when it receives an estimate from the coordinator, every process acks that estimate to all other processes directly, instead of going back to the coordinator; Similarly, when a process suspects the coordinator, it informs all other processes about the suspicion and then moves to the next round.

Each one of those adaptations is discussed below, while the full version of our algorithm is given in Algorithm 1; the corresponding correctness proofs are given in Appendix A. In this algorithm, we construct a local predicate for any other process q at every process p and denoted $Byzantine_p(q)$. Initially this predicate is *false*. As soon as p detects some misbehavior exhibited by q, like sending invalid messages or skipping expected messages, p sets its local predicate $Byzantine_p(q)$ to true.

Echo Broadcast Protocol. To cope with conflicting messages, we use an *echo broadcast* protocol [16, 13]. The echo broadcast forces the coordinator to prove that it does not send conflicting messages to correct processes. The coordinator of some round r, say process q, initiates the echo broadcast by sending its estimate v in an $Initial$ message to all processes. When a correct process p receives this $Initial$ message, it *echoes* v to q by sending it a signed *Echo* message including v; any additional message it receives from q is ignored by p. Once process q receives $N - f$ signed *Echo* messages for v, it sends a *Ready* message that carries the set of received *Echo* messages. No

Algorithm 1 Byzantine-resilient consensus

function *propose*(e_i)
{*Every process* p_i *executes the Task 1 and Task 2 concurrently*}
$InitialCertif_i \leftarrow \emptyset$;
loop { *Task 1*}
  $CurrentRoundTerminated_i \leftarrow false$;
  $ReadyCertif_i \leftarrow \emptyset$; $GoPhase2Certif_i \leftarrow \emptyset$; $DecideCeritf_i \leftarrow \emptyset$;
  $coordSuspect_i \leftarrow false$; $c_i \leftarrow (r_i \ mod \ N) + 1$;

  if $c_i = i$ ***then***
    $send_{all}\ E_i(Initial, p_i, r_i, e_i, InitialCertif_i)$

  while *not* ($CurrentRoundTerminated_i$) ***do***
    when *receive Valid* $E_j(Initial, p_j, e_j, InitialCertif_j)$ *from* p_j
      if $(j = c_i) \wedge$ (*no Echo message was sent in* r_i *by* p_i) ***then***
        send to p_{c_i} $E_i(Echo, p_i, r_i, e_j)$;

    if $c_i = i$ ***then***
      when *receive* $(N - f)$ *messages* $E_j(Echo, p_j, r_i, e)$ *for the same value* e
        $ReadyCertif_i \leftarrow \{E_j(Echo, p_j, r_i, e) : p_i \text{ received } E_j(Echo, p_j, r_i, e) from\ p_j\}$;
        $send_{all}\ E_i(Ready, p_i, r_i, e, ReadyCertif_i)$;

    when *receive Valid* $E_j(Ready, p_j, r_i, e, ReadyCertif_j)$
      $DecideCertif_i \leftarrow DecideCertif_i \cup \{E_j(Ready, p_j, r_i, e, ReadyCertif_j)$;
      if (*First Ready message received*) $\wedge$ ($i \neq c_i$) ***then***
        $ReadyCertif_i \leftarrow ReadyCertif_j$;
        $e_i \leftarrow e$;
        $send_{all}\ E_i(Ready, p_i, r_i, e_i, ReadyCertif_i)$;
      else if ($N - f$ *Ready messages received*) ***then***
        $send_{all}\ E_i(Decide, p_i, r_i, e, DecideCertif_i)$;
        $return(e)$;

      if $(p_{c_i} \in \diamond M_{\mathcal{A}} \vee Byzantine_i(c_i)) \ \wedge (not\ coordSuspected_i)$ ***then***
        $send_{all}\ E_i(Supicion, p_i, r_i)$;
        $coordSuspected_i \leftarrow true$;

    when (*receive* $N - f$ *messages* $E_j(Supicion, p_j, r_i)) \ \wedge \ (phase_i = 1)$
      $GoPhase2Certif_i \leftarrow \{E_j(Supicion, p_j, r_i) : \ p_i \text{ received } E_j(Supicion, p_j, r_i) \text{ from } p_j\}$;
      $send_{all}\ (E_i(GoPhase2, p_i, r_i, e_i, ReadyCertif_i), E_i(GoPhase2Certif_i))$;

    when *receive Valid* $(E_j(GoPhase2, p_j, r_i, e_j, ReadyCertif_j), E_j(SuspectProof_j))$
      if $phase_i = 1$ ***then***
        $phase_i = 2$;
        $InitialCertif_i \leftarrow \emptyset$;
        $send_{all}\ (E_i(GoPhase2, p_j, r_i, e_i, ReadyCertif_i), E_i(SuspectProof_j))$;
      $InitialCertif_i \leftarrow InitialCertif_i \cup (E_j(GoPhase2, p_j, r_i, e_j, ReadyCertif_j)$;
      if ($N - f$ $GoPhase2$ *messages received*) ***then***
        $(e_i, ReadyCertif_i) \leftarrow Last\_Successfully\_EchoBroadcast(InitialCertif_i)$;
        $currentRoundTerminated_i \leftarrow true$;
        $r_i \leftarrow r_i + 1$

when *receive Valid* $E_j(Decide, p_j, r, e, DecideCertif_j)$ {*Task2*}
  $send_{all}\ E_j(Decide, p_j, r, e, DecideCertif_j)$;
  return(e);

correct process p delivers v until it receives the *Ready* message that carries the $N - f$ expected *Echo*.[2] The echo broadcast is illustrated in Figure 5.

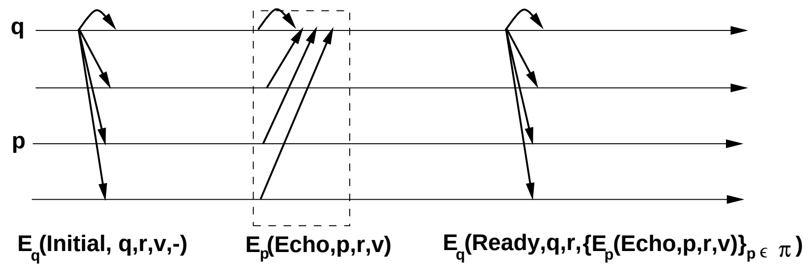

Fig. 5. Echo broadcast protocol

Message Certification. To cope with invalid messages, we introduce the notion of *certificate*. The aim of a certificate is to prove the *validity* of some message $m2$ such that $m1 \prec m2$. Such a validity is proved by exhibiting, in the certificate appended to $m2$, the very fact that the reception of $m1$ has indeed occurred. Then, based on the set associated to $m1$, any correct process can check if $m1$ was correctly taken into account to compute $m2$. Therefore, the structure of any certificate consists in the collection of all signed messages that compose $m1$.[3]
In the following, we detail four kinds of messages in our algorithm that require the use of certificates. The first two are *Initial* and *Ready*: these messages are introduced by the Echo Broadcast protocol. The last two are *Decide* and $GoPhase2$.

- *Initial Message.* In round r, this message, noted here $Initial_r$, carries the estimate proposed by the current coordinator. This message is validated by a certificate named $InitialCertif$.
  For the first round ($r = 1$), since no rounds happened before, $Initial_r$ does not need its certificate to be validated. That is, in this round, the certificate $InitialCertif$ is empty and any estimate proposed by the current coordinator, say q, is considered valid.
  For all other rounds ($r > 1$), the estimate proposed by q must be selected during Phase 2 of round $r - 1$. During this phase, process q should wait for $N - f$ $GoPhase2$ messages, according to which it updates its estimate. We note this set of $(N - f)$ $GoPhase2$ messages $Set\_GoPhase2_{r-1}$. Since $Initial_r$ must be sent after the reception of $Set\_GoPhase2_{r-1}$, we have that $Set\_GoPhase2_{r-1} \prec Initial_r$.
  Consequently, the certificate $InitalCertif$ appended to $Initial_r$ is the collection of all messages that compose $Set\_GoPhase2_{r-1}$. Knowing the update rule that should be processed by q to update its estimate, and having set $Set\_GoPhase2_{r-1}$ that served for this update, each correct process can check the validity of $Initial_r$ message received from q.

- *Ready Message.* In round r, this message, noted here $Ready_r$, is sent by current coordinator in the last step of the Echo Broadcast protocol. This message is validated by a certificate named $ReadyCertif$.
  In round r, the current coordinator should wait for $N-f$ similar *Echo* messages before sending $Ready_r$. We note this set of $(N - f)$ *Echo* messages Set_Echo_r. Since $Ready_r$ must be sent after receiving Set_Echo_r, we have that $Set\_Echo_r \prec Ready_r$. Consequently, the certificate

[2] Lemma 1 in Appendix A shows how this algorithm prevents correct processes from delivering, in the same round, conflicting estimates that may be sent by a Byzantine coordinator.

[3] Theorem 3 in Appendix A shows how the notion of certificate enables to preserve consensus agreement across rounds.

$ReadyCertif$ appended to $Ready_r$ is the collection of all messages that compose Set_Echo_r.[4]

- *GoPhase2 Message.* In round r, this message, noted here $GoPhase2_r$, can be sent by any process q. This message is validated by a certificate named $GoPhase2Certif$.
  In round r, any process should have received at least $N - f$ *Suspicion* messages before sending $GoPhase2_r$. We note this set of $(N - f)$ *Suspicion* messages $Set\_Suspicion_r$. Since $GoPhase2_r$ must be sent after the reception of $Set\_Suspicion_r$, we have that $Set\_Suspicion_r \prec GoPhase2_r$. Consequently, the certificate $GoPhase2Certif$ appended to $GoPhase2_r$ is the collection of all messages that compose $Set\_Suspicion_r$.

- *Decide Message.* In round r, this message, noted here $Decide_r$, can be sent by any process q. This message is validated by a certificate named $DecideCertif$.
  In round r, any process should have received at least $N - f$ $Ready_{r'}$ messages from the same round $r' \leq r$ before sending $Decide_r$. We note this set of $(N - f)$ *Ready* messages $Set\_Ready_{r'}$. Since $Decide_r$ must be sent after the reception of $Set\_Ready_{r'}$, we have that $Set\_Ready_{r'} \prec Decide_r$. Consequently, the certificate $DecideCertif$ appended to $Decide_r$ is the collection of all messages that compose $Set\_Ready_{r'}$.

Detecting Missing Messages. The detection of such misbehaviors is based on the FIFO property of our communication channels and on the following properties of our algorithm:

- In every round r, a correct process p sends only a bounded number of messages to some other process q. We denote this bound by n_{round};

- A message m sent by a correct p while in round r should include the value r.

To illustrate how these two properties are used to detect *Missing* messages, consider the correct process p waiting, in round r, for some specific message m' from process q:

- As soon as p has received a message from q with round number $r' > r$, process p detects that q has skipped the expected message;

- As soon as p has received more than $r \cdot n_{round}$ messages from q, process p detects that q has skipped the expected message.

5 Failure Transparency

In the crash failure model, building a consensus black-box on top of failure detectors has the advantage of fully encapsulating all failure-related issues and providing a powerful abstraction for solving other agreement problems [1, **?**,**?**]. So, for an agreement algorithm that is built on top of consensus, failure management is completely invisible (see Figure 6.a). We call this nice aspect *failure transparency.* Achieving the same level of failure transparency in a Byzantine environment is not possible, but again there is room for an intermediate approach. We illustrate here this issue on a well-known agreement problem, namely, the atomic broadcast problem that was solved in the crash model by a simple and elegant reduction to the consensus [1], i.e., it is built using consensus as an underlying building block.

5.1 Reducing Atomic Broadcast to Consensus

In atomic broadcast, correct processes have to agree on the delivery order of messages and every message broadcast by a correct process should eventually be delivered by every correct process. T can be very useful for instance to implement a replicated service [**?**]; the specification of atomic

[4] When a coordinator builds this certificate, we say that it has *successfully echo certified* its estimate.

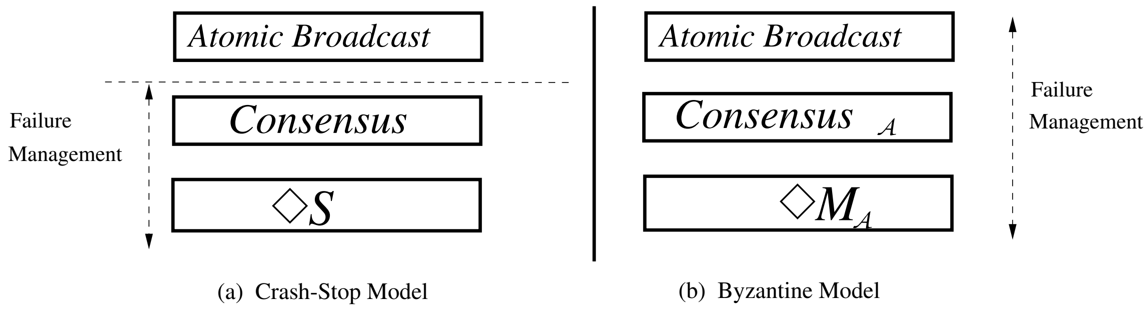

Fig. 6. Failure transparency

broadcast is recalled in Appendix B. In [1] an atomic broadcast algorithm is built on top of a consensus black-box, i.e., atomic broadcast is *reduced* to consensus. Roughly speaking, the reduction algorithm proposed in [1] aims at transforming atomic broadcast into a sequence of consensus instances. Each consensus instance is responsible for atomically delivering a batch of messages.

Now, let us assume that the consensus black-box used for the reduction is resilient to Byzantine failures (e.g., our algorithm proposed in the previous section). Does the reduction of [1] still works in the Byzantine model? The answer is *no*.

In a consensus algorithm, nothing prevents the processes from deciding on a corrupted value proposed by a Byzantine process. In fact, the decided value has only to be a value proposed by **some** process (not necessary a correct one). As a consequence, when executing a sequence of consensus, each one may return a corrupted batch proposed by some Byzantine process. This can lead correct processes to always decide on batches containing only corrupted messages. In others words, with such a non-restrictive consensus validity, the validity of the atomic broadcast can be violated, i.e., messages from correct processes might never be delivered. In fact, the Byzantine behaviors may appear at every level of our stack of algorithms (see Figure 6.b).

5.2 Atomic Broadcast in a Byzantine Model

In the following, we propose a wy to adapt the reduction of [1] to the Byzantine model, by incorporating some of the failure management in the atomic broadcast algorithm itself, i.e., outside of the consensus box. As in [1], we also use an underlying *reliable broadcast* algorithm, used by correct processes to send and receive messages. This algorithm provides two primitives *R_broadcast* and *R_deliver* that allow correct processes to broadcast and deliver messages according to the semantics of the reliable broadcast; the specification of reliable broadcast is recalled in Appendix B.

Reduction Algorithm. The full version of our reduction algorithm is given in Algorithm 2, while the correctness proofs are given in Appendix B. The atomic broadcast interface is defined by two primitives: *A_broadcast* and *A_deliver*, which allow correct processes to broadcast and deliver messages according to the semantics of the atomic broadcast.

Let us now see more precisely how this algorithm proceeds. First, an *A_broadcast* of a message m is performed through $Task$ 1, by invoking the primitive *R_broadcast* with message m. Second, a correct process *A_deliver*s the received messages through $Task$ 2 and $Task$ 3. These tasks manipulate three sets of messages:

- *R_delivered*. This set contains the messages *R_deliver*ed to process p. So, in $Task$ 2, each time a correct process p *R_deliver*s a message m, it inserts m into *R_delivered* set.

- *A_delivered*. This set contains messages that have been *A_deliver*ed to process p.

- *A_undelivered*. This set contains the messages that have been *R_deliver*ed, but not yet *A_deliver*ed.

In $Task$ 3, when a correct process p notices that the set *A_undelivered* is not empty, it starts constructing a vector of propositions with which it launches a new instance of consensus. Let k be a sequence number that disambiguates between different concurrent executions of consensus instances. Before launching the k^{th} consensus instance, each correct process p sends to all processes a signed message $(p, k, A\_undelivered)$. Then, p waits for $f + 1$ signed messages of the form $(q, k, A\_undelivered_q)$ from different processes. These messages are collected in a vector $VectPropose$, where $VectPropose[q] = (q, k, A\_undelivered_q)$ if p received a message from q, and $VectPropose[q] = null$ otherwise. Once the construction of $VectPropose$ is completed, p launches the k^{th} consensus instance with $VectPropose$ as its initial value.

The goal here is to make sure that any initial value proposed by some process (correct or not) is of the form $VectPropose$: for the k^{th} consensus instance, the initial value of any process should be a vector of $f+1$ signed messages of the form $(q, k, A\_undelivered_q)$. Then, if a Byzantine process proposes an initial value that does not respect this format, its value is considered to be corrupted and is not accepted by correct processes. Thanks to the required $VectPropose$ format, we ensure that, in the presence of at most f Byzantine processes, any decided value contains at least one batch of messages proposed by a correct process.

Finally, once the k^{th} consensus instance has been launched, process p waits for the k^{th} decision, which is carried by $VectDecide$. Having the decision vector $VectDecide$, each correct process takes the union of all messages that appear in this vector, except the ones that have been already *A_deliver*ed, and then delivers them in some deterministic order.

Algorithm 2 Byzantine-resilient atomic broadcast with consensus

{*Every correct process p executes the following*}
Initialization
$R\_delivered \leftarrow \emptyset$
$A\_delivered \leftarrow \emptyset$
$k \leftarrow 0$
{*Task1, Task2 and Task3 can be executed concurrently*}
procedure *A_broadcast* {*Task 1*}
  $R\_broadcast(m)$

procedure *A_deliver*
  when *R_deliver(m)* {*Task2*}
    $R\_delivered \leftarrow R\_delivered \cup \{m\}$
  when $R\_delivered - A\_delivered \neq \emptyset$ {*Task3*}
    $k \leftarrow k + 1$
    $A\_undelivered \leftarrow R\_delivered - A\_delivered$
    for $i = 1$ *to* $i = N$ ***do***
      $VectPropose[i] \leftarrow null$
    $send_{all} E_p(p, k, A\_undelivered)$
    when *received* $(f + 1)$ $E_q(q, k, A\_undelivered_q)$ *from different processes*
      $VectPropose[q] \leftarrow E_q(q, k, A\_undelivered_q)$
    $propose(k, VectPropose)$
    wait until $decide(k, VectDecide)$
    $A\_deliver^k \leftarrow \cup_{l=1}^{l=f+1} VectDecide[l] - A\_delivered$
    atomically deliver all messages in $A\_deliver^k$ *in some deterministic order*
    $A\_delivered \leftarrow A\_delivered \cup A\_deliver^k$

6 Related Work

Beside our work, we know of two research efforts that aimed at extending the notion of crash 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 [11]. They defined the notion of *quiet process*, which is close to our notion of muteness. Based on this notion, they introduced a failure detector, noted $\diamond\mathcal{S}$ (bz), which satisfies *Strong Completeness* and *Eventual Weak Accuracy* properties. The aim of $\diamond\mathcal{S}$ (bz) is to track processes that prevent the progress of the algorithm using it.

In [9], Kihlstrom et al. defined the Eventual Weak Byzantine failure detector $\diamond\mathcal{W}$ (Byz). Contrary to the specification of [11], and contrary to the one we propose, this class of failure detectors captures all *detectable* Byzantine behaviors.[5] The set of failures captured by $\diamond\mathcal{W}$ (Byz) include those that prevent the progress of the algorithm, i.e., muteness.

None of [11] or [9] gave a precise definition of their failure detectors. Consequently, they did not have to face the issue related to the *non-orthogonality* of the specification (Section 2). This issue obviously applies to their work as well. Because none of them tried to provide an implementation of failure detectors in a Byzantine environment and prove it correct, they did not have to face the *non-generality implementation* issue either (Section 3). However, without restricting the class of algorithms that can make use of such failure detectors, we can easily end up with contradicting failure detectors properties, as we have pointed out in this paper. In fact, in [9], the authors presented a failure detector implementation that roughly aims at inserting a timeout mechanism in the branch of the algorithm where there are some expected messages. None of those papers has exhaustively addressed the issue of approximating an interval between algorithmic messages, which is a complex task that does not depend only on the synchrony assumptions of the system.

Both [11] and [9] described also consensus algorithms in a Byzantine environment. The aim of our approach was not to describe a consensus algorithm that is more efficient or more resilient than previous ones, but rather to focus on the modularization of such algorithms, and hence address the more general problem of *failure encapsulation* (Section 4). Similarly, several byzantine resilient atomic broadcast algorithms have been proposed [13, 8, 12]. By trying to follow the original reduction of [1], we suggested a simple and more modular approach that illustrates the issue of *failure transparency* (Section 5).

7 Conclusion

Devising distributed algorithms is difficult because of the possibility of partial failures. To help factorize this difficulty, Chandra and Toueg suggested a modular approach where failure detection is encapsulated inside a specific oracle called a *failure detector* [1]. They have shown how this modular approach simplifies the reasoning about the correctness of agreement algorithms.

Since the original failure detector approach was proposed in the context of crash-failures, a natural question comes to mind: does it make sense to follow a similar approach in the context of Byzantine failures? In other words, is there a notion of *Byzantine failure detector* that also enables a nice modularization?

To answer this question, we have considered four nice modularity aspects of crash failure detectors, and we have argued that none of those aspects can be completely kept as it is in a Byzantine context. The fundamental reason is the inherent nature of Byzantine failures: it is just impossible to completely separate Byzantine failure detectors from their context, i.e., from the algorithms using failure detectors. Roughly speaking, this is because the very definition of a Byzantine failure is itself context sensitive: a process commits a Byzantine failure if it deviates from its *specification*. Simply giving up the failure detector approach in a Byzantine context would however not be very constructive, because an algorithm designer would be left with the alternative of mixing failure detection with all other aspects of distributed processing in an ad hoc manner.

[5] These include all failures, except those related to the application, such as cheating on an initial value.

We have suggested an intermediate approach that preserves part of the modularity of the original failure detector approach. Other intermediate approaches might be possible and the objective of this paper was not to advocate one approach in particular. As we pointed out however, coming out with a sound failure detector notion in a Byzantine environment is fraught with the danger of proposing specifications that are, either useless, or impossible to implement even in completely synchronous systems.

Acknowledgments

We are very grateful to Jean-Michel Helary and Mike Reiter for their constructive comments on this work, and to Sidath Handurukande and Corine Hari for their help in improving the presentation of this paper.

References

1. T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.
2. 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.
3. C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, apr 1988.
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 *Proceedings of the International Conference on Foundations of Computations Theory*, pages 127–140, Borgholm, Sweden, 1983.
6. R. 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.
7. 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.
8. 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.
9. 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.
10. L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.
11. D. Malkhi and M. Reiter. Unreliable Intrusion Detection in Distributed Computations. In *Proceedings 10th Computer Security Foundations Workshop (CSFW97)*, pages 116–124, June 1997.
12. L. E. Moser, P. M. Melliar-Smith, and D. A. Agrawala. Total ordering algorithms. In *ACMCSC: ACM Annual Computer Science Conference*, 1991.
13. M. K. Reiter. Secure Agreement Protocols: Reliable and Atomic Group Multicast in Rampart. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pages 68–80, November 1994.
14. 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.
15. A. Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3):149–157, April 1997.
16. S. Toueg. Randomized Byzantine Agreements. In *Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing*, pages 163–178, August 1983.

Appendix A: Correctness of the Consensus Algorithm

We prove in this section that Algorithm 1 solves consensus in the presence of less than one third of Byzantine processes (the rest must be correct) using failure detector $\diamond M_{\mathcal{A}}$. We first recall below the properties of consensus and detail the assumptions underlying our proofs.

Consensus Problem Specification. The problem consists for a set of processes to *decide* on some common value among a set of *proposed* values [1]. The agreement must be reached despite the presence of Byzantine processes. Formally, the processes need to satisfy the following properties:

- **Validity:** if a correct process decides v then v was proposed by some process.
- **Agreement:** no two correct processes decide differently.
- **Termination:** every correct process eventually decides.

System Assumptions. The system is composed of a finite set $\Omega = \{p_1, ..., p_N\}$ of N processes, fully interconnected through a set of reliable FIFO communication channels. The maximum number f of Byzantine processes is less than $N/3$. 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 $\mathbb{N}$. It is important to notice that this clock is not accessible to processes: it is solely used to reason about runs. The timing assumptions that are accessible to the processes are encapsulated within $\diamond M_{\mathcal{A}}$. We assume that the channels and the messages are authenticated. Any correct process can authenticate the sender of each message, and messages have signatures such as RSA [14]. Roughly speaking, the authentication prevents an Byzantine process from changing the contain of some message it relays, or sending some message and claiming to have received this message from another process. In the following, the context of every lemma and theorem is Algorithm 1.

Lemma 1 *In any round r, the coordinator (correct or not) of r cannot successfully echo certify two different values v and w.*

PROOF: The proof is by contradiction. Assume that in round r the coordinator, say process p, successfully echo certified two value v and w. Consequently, process p constructs: (1) a *Ready* message for v, with a *ReadyCertif* composed of $N - f$ messages $E_j(Echo, p_j, r, v)$, and (2) a *Ready* message for w, with *ReadyCertif* composed of $N - f$ messages $E_j(Echo, p_j, r, w)$. Therefore, in round r, two sets of $N - f$ processes sent *echo* messages for two distinct values v and w. In presence of less than $N/3$ Byzantine processes, the previous statement is correct only if at least one correct process sent two conflicting *echo* messages. Since, each correct process sends at most one *echo* message, then no two different messages can be successfully echo certified in the same round: a contradiction. □

Lemma 2 *Let r be the smallest round for which at least $f + 1$ correct processes deliver an estimate v that was successfully echo certified. Then, no other value $w \neq v$ can be successfully echo broadcast in any subsequent round r' (i.e., $r' \geq r$).*

PROOF: The proof is split in two cases.

Case 1 ($r = r'$) : Immediat from Lemma 1.

Case 2 ($r' > r$): The proof is by contradiction. Assume that l is the first round for which the lemma does not hold. Thus, at least one correct process receives, in round l, a valid *Ready* message that carries an estimate w different than v. A valid *Ready* message implies that the appended certificate *ReadyCertif* is a set of $N - f$ messages $E_q(Echo, q, l, w)$. So, at least $f + 1$ correct processes sent an *Echo* of w to the coordinator of round l, say process p. Therefore, at least $f + 1$ correct processes find the message $Initial$, sent by process p and carrying w, valid. A valid $Initial$ message implies that the appended certificate $InitialCertif$ is a set of $N - f$ messages $E_q(GoPhase2, q, l-1, e_q, ReadyCertif_q)$ such that, among these $N - f$ messages, w corresponds to the last value successfully echo broadcast. On the other hand, by assumption we know that the lemma holds for all rounds k such that $r \leq k < l$. Consequently, until round $l-1$, no value different than v was successfully echo certified, and there is at least $f+1$ correct processes that delivered v. To construct its certificate $InitialCertif_p$, the coordinator p of round l needs the participation of at least $N - f$ processes that sent to p their $GoPhase2$ message from round $l - 1$. Therefore, there is at least one correct process q that has delivered v in round $l - 1$, which has its $GoPhase2$ in $IntialCertif_p$. Therefore, the last value successfully echo certified that appears in $InitalCertif_p$ is v. Thus w could not be validated by $InitialCertif$ of p: a contradiction. □

Theorem 3 *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 values, v and w respectively, during rounds r and r' such that $r \leq r'$. In order to decide, each correct process must receive at least $N - f$ *Ready* messages for the same estimate (if it either decides in Task 1 or Task 2). Therefore, when p decides on v, in round r , there is at least $f + 1$ correct processes that deliver the same value v in some round $k \leq r$. Consequently, from Lemma 2, there is no other value w different than v that could be successfully echo certified in any round $r' \geq k$. Consequently, the correct process q could not decide on some $w \neq v$. This contradicts our assumption, i.e., q decides $w \neq v$. □

Lemma 4 *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 5 *If no correct process decides in round r, then 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 $N - f$ valid *Ready* messages. Thus eventually some correct process receives at least $N - f$ 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 $\mathcal{A}$-completeness of $\diamond M_{\mathcal{A}}$ process p eventually suspects p_c, or (2) p_c sends a non valid *Ready* message 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 $N - f$ *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 $GoPhase2$. 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 $GoPhase2$ message to all. Thus each correct process receives at least $N - f$ messages $GoPhase2$. This allows Phase 2 of round r to be completed. Consequently, each correct process proceeds to next round, i.e., $r + 1$. □

Theorem 6 *Eventually every correct process decides (Termination).*

PROOF: By the eventual weak $\mathcal{A}$-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 5, one correct process has decided in a round $r' < r$, and so, by Lemma 4 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 $N - f$ messages *Echo* message needed to successfully echo certified its estimates. By the reliable channel assumption, 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 *The decided value is a value proposed by some process (Validity).*

PROOF: Trivially ensured since any decided value is some value proposed by a coordinator, i.e., some process. □

Appendix B: Correctness of the Atomic Broadcast Algorithm

We prove in the following that Algorithm 2 solves the atomic broadcast problem in the presence of Byzantine failures and two third of correct processes. We make here the same assumptions as in Appendix A. Algorithm 2 relies in addition on the properties of both consensus and reliable broadcast. The reliable broadcast specification is given below:

– **Agreement:** if a correct process *R_deliver*s a message m, then all correct processes eventually *R_deliver* m.
– **Validity:** if a correct process *R_broadcast*s a message m, then it eventually *R_deliver*s m.
– **Integrity:** for any message m, every correct process *R_deliver*s m at most once, and if the sender of m is some correct process p then m was previously *R_broadcast* by p.

Before going into the proofs, we first recall the specification of atomic broadcast.

Atomic Broadcast Specification. We define the interface of the atomic broadcast by two primitives: *A_broadcast* and *A_deliver*, which allow correct processes to broadcast and deliver messages according to the semantics of the atomic broadcast. This semantics guarantees that (1) all correct processes deliver the same set of messages (Agreement), (2) all correct processes deliver messages in the same order (Total order), (3) the set of delivered messages includes all the messages broadcast by correct processes (Validity), and (4) no spurious messages are delivered from correct processes (Integrity). The intuitive properties presented above can be formalized as follows:

– **Agreement:** if a correct process *A_deliver*s a message m, then all correct processes eventually *A_deliver* m.
– **Total order:** if two correct processes p and q *A_deliver* two messages m and m', then p *A_deliver*s m before m' if and only if q *A_deliver*s m before m'.
– **Validity:** if a correct process *A_broadcast*s a message m, then it eventually *A_deliver*s m.
– **Integrity:** for any message m, every correct process *A_deliver*s m at most once, and if the sender of m is some correct process p then m was previously *A_broadcast* by p.

In the following, the context of every lemma and theorem is Algorithm 2.

Lemma 8 *Consider any two correct processes p and q, and any message m, if $m \in R\_delivered_p$, then we eventually have $m \in R\_delivered_q$.*

Proof: Trivially follows from the agreement property of reliable broadcast. □

Lemma 9 *For any two correct processes p and q, and any $k \geq 1$:*

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 executes $propose(1, *)$ then q 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 8, m is eventually in $R\_delivered_q$ for any correct process q. Since $A\_delivered_q$ is initially empty for any process q, $R\_delivered_q - A\_delivered_q \neq \emptyset$ eventually holds. So, each process q eventually executes $send_{all}E_q(q, 1, *)$. in the presence of at most f incorrect processes, each process q constructs its vector $VectPropose$ and hence executes $propose(1, *)$.

Lemma part 2, $k = 1$: we prove that if p $A_deliver$s a messages in $A\_deliver_p^1$, then q eventually $A_deliver$s messages in $A\_deliver_q^1$ such that $A\_deliver_p^1 = A\_deliver_q^1$. If p $A_deliver$s messages in $A\_deliver_p^1$, then it has previously executed $propose(1, *)$. By part 1 of Lemma 9, each correct process eventually executes $propose(1, *)$. By termination, validity and agreement properties of consensus, all correct processes eventually decide on the same vector of messages $VectDecide^1$. Since $A\_delivered_p^1$ and $A\_delivered_q^1$ are initially empty, and $VectDecide_p^1 = VectDecide_q^1$ then $A\_deliver_p^1 = A\_deliver_q^1$.

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 executes $propose(n, *)$ then eventually q 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 8 m is eventually in $R\_delivered_q$. Therefore, for any process q $R\_delivered_q - A\_delivered_q \neq \emptyset$ eventually holds. So, each process q eventually executes $send_{all}E_q(q, n, *)$. In presence of at most f Byzantine processes, each process q constructs its vector $VectPropose$ and hence executes $propose(n, *)$.

Lemma part 2, $k = n$: We prove that if p $A_deliver$s messages in $A\_deliver_p^n$ then q eventually $A_deliver$s messages in $A\_deliver_q^n$ such that $A\_deliver_p^n = A\_deliver_q^n$. If p $A_deliver$s messages in $A\_deliver_p^n$ then, p has previously executes $propose(n, *)$. By part 1 of this lemma each correct process executes $propose(n, *)$. By termination, validity and agreement properties of consensus, all correct processes eventually decide on the same vector of messages $VectDecide^n$. Consequently, we have $VectDecide_p^n = VectDecide_q^n$. By the induction hypothesis for all $1 \leq k < n$ we have $A\_deliver_p^k = A\_deliver_q^k$. Hence, the set of messages $A\_deliver_p^n$ delivered by p is equal to the set of messages $A\_deliver_q^n$ delivered by q.

□

Theorem 10 *All correct processes deliver the same set of messages in the same order (agreement and total order).*

Proof: Immediate from Lemma 9 and the fact that all correct processes deliver the messages in the same deterministic order. □

Theorem 11 *If some correct process $A\_broadcasts$ a message m, then it eventually $A\_delivers$ m (validity).*

Proof: The proof is by contradiction. We assume that a correct process p $A_broadcast$s a message m and never $A_deliver$s m. Thus, by Lemma 10 no correct process $A_deliver$s m.

By Task 1 of Algorithm 2, we know that to $A\_broadcast$ a message m a correct process p executes $R\_broadcast(m)$. Therefore, thanks to validity and agreement properties of reliable broadcast, eventually every correct process q $R_deliver$s m in Task 2, i.e., eventually $m \in R\_delivered_q$. on the other hand, by assumption we state that no correct process q never $A_deliver$s m, so no process q inserts m in $A\_delivered_q$.

From Algorithm 2, there exist a constant k, such that for all $l \geq k$, every correct process q has $m \in R\_delivered_q^l - A\_delivered_q^l$, i.e., $m \in A\_undelivered_q^l$. Then, each process q executes $send_{all}E_q(l, q, A\_undelivered_q)$ and builds its $VectPropose$ with which it launches its l^{th} consensus. By termination and agreement properties of consensus, we infer that all correct processes eventually decide on the same $VectDecide^l$. Furthermore, by our restriction in Algorithm 2 on initial values format and the validity property of consensus, we ensure that any decision $VectDecide^l$ returned by l^{th} consensus is a vector of $(f + 1)$ signed messages $E_s(s, l, A\_undelivered_s)$ from different processes. So, in the presence of at most f Byzantine processes, there is at least one message $E_s(s, l, A\_undelivered_s)$ from some correct process s. Consequently, we have $m \in A\_undelivered_s$. Since the decision is on the union of all messages in $VectDecide^l$ except those which are already $A_deliver$ed, m is eventually $A_deliver$ed by all correct processes: a contradiction. □

Theorem 12 *For any message m, every correct process $A\_delivers$ m at most once, and if the sender of m is a correct process p, then m was previously $A\_broadcast$ by p (integrity).*

Proof: Suppose that a correct process p $A_deliver$s a message m, p inserts m in its set of delivered messages $A\_delivered_p$. So, from Algorithm 2 p delivers m at most once.

Now, if a correct process p $A_deliver$s a message m then, it has previously decided on $VectDecide^k$ for some k, such that $m \in VectDecide^k$. By the restriction imposed in Algorithm 2 on initial values format and the validity property of consensus, we ensure that $VectDecide^k$ is composed of $f + 1$ signed messages $E_q(k, q, A\_undelivered)$. If m belongs to a batch of messages $A\_undelivered_q$ such that q is Byzantine, we cannot infer that some processes has effectively executed $A\_broadcast(m)$. Otherwise, i.e., if m belongs to a batch $A\_undelivered_q$ such that q is correct then we infer that q has executed in Task 1 $R\_deliver(m)$. Then, by the integrity property of reliable broadcast, if the sender of m, say p, is a correct process, then p $R_broadcast$s message m, which means that p has $A\_Broadcast$ message m in Task 1. □