CHAPTER 2

Tolerating Arbitrary Failures With State Machine Replication

ASSIA DOUDOU[1], BENOÎT GARBINATO[2], and RACHID GUERRAOUI[1,*]

[1]Swiss Federal Institute of Technology in Lausanne, EPFL; and [2]University of Lausanne, Switzerland

2.1 INTRODUCTION

2.1.1 Motivation

The growing reliance, in our daily lives, on services provided by distributed applications (e.g., air-traffic control, public switched telephone networks, electronic commerce, etc.) renders us vulnerable to the failures of these services. The challenge of fault tolerance consists in providing services that survive to the occurrence of failures.

The design and verification of fault-tolerant distributed applications is however viewed as a difficult task. In recent years, several paradigms have fortunately been identified which simplify this task. Key among these paradigms is *state machine replication* [12, 15, 19]. The underlying idea is intuitively simple. In short, every crucial service that needs to be made fault tolerant is replicated on several computers that are supposed to fail independently. The presence of several replicas ensures the high availability of the service. To preserve the consistency of the service, invocations of its replicas, even if coming from different clients, are then handled in such a way that they reach the replicas in the same order. The abstraction that provides this guarantee is called the *total order broadcast* primitive. Roughly speaking, this communication primitive ensures that messages broadcast within a group of processes are delivered in the same order, despite concurrency and failures.

*Contact author: R. Guerraoui, LPD EPFL CH 1015 Lausanne, Switzerland (E-mail: Rachid.Guerraoui@epfl.ch).

Dependable Computing Systems. Edited by Hassan B. Diab and Albert Y. Zomaya
ISBN 0-471-67422-2 Copyright © 2005 John Wiley & Sons, Inc.

Implementing this abstraction is however very challenging. The motivation of this work is precisely to address that challenge and to identify a set of abstractions to build upon a total order broadcast algorithm that tolerates (1) *Byzantine (arbitrary)* failures of processes and (2) *asynchronous* periods of the network (i.e., network failures). (1) Basically, a Byzantine process may arbitrarily deviate from the specification of its algorithm: intuitively, it can be *malicious* and act *against* the algorithm assigned to it. Byzantine failures can be the most disruptive, and there is anecdotal evidence that such failures do occur in practice. (2) Tolerating asynchronous periods means in our context that messages should not be delivered out of order by different processes during periods where the system is overloaded and synchronous delays are not respected. Algorithms that tolerate such asynchronous periods are sometimes called *indulgent* [11]. Being able to tolerate Byzantine failures in an indulgent way is particularly important because an attacker can delay the communication between nonfaulty processes during an arbitrary period of time.

2.1.2 Background

Our work follows the modular approach of Chandra and Toueg [5]. In their seminal paper, Chandra and Toueg presented a total order broadcast algorithm based on *reliable broadcast* and *consensus* modules. Consensus is itself based on a *failure detector* module, together with *reliable communication channels*. Their layered architecture has two appealing properties. On the one hand, it enables the better understanding, verification, and implementation of the total order broadcast algorithm. For example, one can optimize any of the two underlying modules, without changing the total order broadcast algorithm, as long as the specifications of these modules do not change. On the other hand, it helps clearly identify the assumptions under which the safety and liveness properties of the algorithm are ensured. In particular, encapsulating synchrony assumptions underlying the (unreliable) failure detector module enables to clearly show that safety does not rely on such assumptions, i.e., the ordering of messages is not violated even during asynchronous periods of the network.

The approach of Chandra and Toueg was however devised for a crash-stop model and adopting a similar approach in a Byzantine environment is rather challenging. To see why, consider (1) their total order broadcast algorithm using consensus and (2) their consensus algorithm using a failure detector.

1. The total order broadcast algorithm consists of a sequence of consensus instances, each used to agree on a batch of messages to deliver. In a Byzantine environment, an attacker can keep on proposing the same set of messages creating the possibility for correct processes to keep on indefinitely deciding on that very same set of possibly bogus messages, i.e., messages that no correct process has ever broadcast. Consequently, messages broadcast from correct processes might never be delivered by any correct process.
2. The consensus algorithm relies on a failure detector module that provides hints about crashed processes. In particular, this module prevents processes

from indefinitely waiting for messages from crashed processes. Even if we assume a perfect failure detector that accurately detects crashed processes [5], Byzantine processes might, without crashing, stop sending messages and hence block correct processes. In fact, as we discuss in the paper (Section 2.6), it is easy to see that the original notion of failure detector is meaningless in a Byzantine environment.

Clearly, one cannot build a Byzantine-resilient total order broadcast algorithm using the abstractions of [5], i.e., simply by changing the implementations of those abstractions. The challenge here is to identify alternative abstractions that fit a Byzantine model and devise algorithms that implement these abstractions under realistic assumptions.

2.1.3 Contribution

This paper presents a total order broadcast algorithm based on a *weak interactive consistency* and a *reliable broadcast* modules. The latter is itself based on a *muteness failure detector* module together with *reliable communication channels*.

Weak interactive consistency is an agreement abstraction. As its name indicates, this abstraction is weaker than the traditional interactive consistency abstraction [9]. Whereas the latter does not have any solution that tolerates asynchronous periods of the system, the former does. It is in this sense close to the traditional consensus abstraction. However, unlike consensus, where processes need to agree on exactly one value, the processes need here to agree on a set of values (that contains at least one value from a correct process): this difference is precisely what makes it an adequate building block to simplify the task of ordering messages in a Byzantine environment.

We give a weak interactive consistency algorithm that tolerates Byzantine failures and asynchronous periods using a *muteness failure detector* module. Roughly speaking, this module generalizes the notion of crash failure detector to *muteness* failures. These constitute a subset of Byzantine failures, which are induced by processes from which correct processes stop receiving algorithmic messages. A major challenge here is to precisely capture the relative flavour of this notion: muteness is inherently dependent on a given algorithm. We define our muteness failure detector module in terms of abstract axiomatic properties and we then address the implementability of this module in a partial synchrony model, i.e., basically, assuming that the system can alternate between asynchronous and synchronous periods that last for long enough for the algorithm to terminate.

Building our total order broadcast algorithm using a layered architecture enables us to reason precisely about the correctness of our algorithm. In particular, we identify the assumptions under which the liveness of our algorithm is ensured and express them in terms of precise properties of a muteness failure detector. As for safety, our total order algorithm always preserves it (even during asynchronous periods), provided that the maximum number of Byzantine processes is less than one-third of the total number of processes. Our layered architecture also enables us to optimize the algorithm in a modular way. We illustrate this by showing how our algorithm can easily be optimized for steady state (nice periods of the system where the system is synchronous and no failures occur, i.e., the most frequent in practice). We obtain a communication pattern in steady state that is similar to that of a decentralized 3PC [20]: three communication steps are sufficient for a message to be delivered. Furthermore, our weak interactive consistency abstraction inherently enables the grouping of messages and their delivery in the forms of batches (just like a group commit in transactional systems [10]).

2.1.4 Related Work

The idea of devising Byzantine-resilient group communication primitives (e.g., total order broadcast) was pioneered in Rampart [17] and then further explored in SecureRing [14]. In both systems, total-order broadcast algorithms rely on an underlying membership service. When a process is suspected to be faulty (even when the suspicion is wrong and the process is just slow), a view change is triggered and the suspected process is excluded from the new view. Preserving safety becomes problematic because the assumption that a maximum number of Byzantine processes is less than one-third of the total number of processes (i.e., in a given view) may not be true after several exclusions. In our case, the algorithm is intrinsically aware that suspicions might be wrong, and false suspicions are not turned into accurate ones by excluding suspected processes. In short, Rampart [17] and SecureRing [14] assume a reliable failure detection scheme, whereas we assume an eventually reliable failure detection scheme: synchrony assumptions are only required to hold eventually.

The approach of that described in Castro and Liskov [3] is closer to ours in the sense that it also requires synchrony assumptions only eventually and only for liveness. The proposed message-ordering scheme is based on the idea of sequenced views but with *no exclusion of processes.* In each view, only one process (the primary) is responsible for ordering client requests. Contrary to our approach, however, the approach of Castro and Liskov [3] is monolithic. Agreement issues are not encapsulated within any abstraction (e.g., weak interactive consistency) and, more importantly, liveness requirements are not expressed through any abstract axiomatic property (e.g., some form of muteness failure detector properties). Instead, liveness relies on the use of time outs (scattered throughout the algorithm) to prevent correct processes from being blocked by a Byzantine primary and on the following low-level assumption on the sequencing of views: “there eventually is a view with a correct primary that other correct processes will not time out.” As we discuss in the paper, expressing liveness requirements in an abstract and precise way is not trivial. Interestingly, and even if the ordering scheme of the previous study [3] was primarily devised with performance in minds (i.e., not modularity), we argue in Section 2.6 that our modular approach promotes effective and systematic optimizations that enables us to reach comparable performance. In steady state, just like ours, the communication pattern of the previous study [3] is that of a decentralized 3PC [20].

As we pointed out, our scheme inherently gathers messages that can be delivered in the form of batches. Hence, rather than hampering optimizations, our modularization actually promotes them.

In Baldoni et al. [1], issues of moving from a crash-resilient to a Byzantine-resilient consensus algorithm are considered and addressed in a systematic way. Many of those issues are similar to those we address in this paper except for the case of consensus. In particular, the authors make use of our muteness failure detection abstraction as a central mechanism to ensure the liveness of their consensus algorithm. Moreover, weak-interactive consistency can be viewed as an answer to the question opened by Baldoni et al. [1] about identifying a meaningful consensus-like abstraction for a Byzantine context, e.g., weak-interactive consistency.

2.1.5 Roadmap

The rest of this chapter is organized as follows. Section 2.2 presents our system model. Section 2.3 gives the specification of total order broadcast and presents an algorithm that implements it using a reliable broadcast abstraction and our weak interactive consistency abstraction. Section 2.4 describes how to implement weak interactive consistency using muteness failure detectors. Section 2.5 discusses the specification and implementation of these failure detectors. Section 2.6 concludes Chapter 2 with a discussion about the modularity of our approach. For space limitation, most correctness proofs have been omitted: they can be found in Ref. [6].

2.2 SYSTEM MODEL

2.2.1 Execution and Communication Model

We consider a distributed system composed of a set $\Pi = \{p_1, p_2, \ldots, p_N\}$ of N processes. Processes communicate by message passing via a fully connected network composed of reliable point-to-point channels; i.e., if process p sends message m to process q, and both q and p are correct, then q eventually receives m. These channels are supposed to guarantee FIFO order and preserve the integrity of messages exchanged between correct processes; in particular, the content of messages cannot be altered by some active intruder. Every message m contains a field $seq(m)$ such that $seq(m)$ is a unique identifier composed of the sender's identity $seq.id(m)$ and of a sequence number, e.g., the local clock of the sender.

2.2.2 Byzantine Failure Model

We assume a *Byzantine failure model with message authentication* [16]. In such a model, either a process is correct and it executes the algorithm assigned to it, or it is Byzantine. In this case, the processes can fail by crashing (i.e., prematurely stop participating in the protocol) but can also behave maliciously. A Byzantine (we also say faulty or incorrect) process can for instance send garbled and misleading messages or can refuse to send expected messages. More generally, *Byzantine processes*, also known as *malicious processes*, can exhibit arbitrary behaviors. In contrast, a correct process executes an infinite number of steps and respects the specification of the algorithm assigned to it.

There exist however limits to the power of malicious processes: thanks to message authentication, a malicious process cannot impersonate correct processes. Message authentication relies on the following *signature unforgeability assumption*: if a correct process p does not send a signed message m, then no correct process ever receives a message m signed by p. We assume that every correct process signs each of its messages before sending it.[1] In addition, we assume that the maximum number of Byzantine processes among the N servers is $f < N/3$. There is no restriction on the number of Byzantine clients.

2.3 TOTAL ORDER BROADCAST

We first give below the properties of total order broadcast, then we describe the underlying abstractions, and finally we present the actual algorithm making use of those abstractions.

2.3.1 Specification

Total order broadcast is defined through two primitives: A_deliver and A_broadcast. The semantics of these primitives are captured by the following properties:

- **Validity**. If a correct process A_broadcasts a message m, then eventually some correct process A_delivers m.
- **Agreement**. If a correct process A_delivers some message m, then eventually all correct processes A_deliver m.
- **Integrity**. For every message m, every correct process p A_delivers m at most once. Furthermore, if the sender of m, say q, is correct, then no correct process A_delivers m unless q has previously A_broadcast m.
- **Total order**. If two correct processes p and q, both A_deliver two messages m and m', then p A_delivers m before m' iff q A_delivers m before m'.

Basically, all messages broadcast by a correct process are eventually delivered by all correct processes (validity), all correct processes agree on the set of messages they deliver (agreement), no spurious message is ever associated to a correct process (integrity), and all correct processes additionally agree on the delivery order of messages (total order). The specification that we consider is similar to the traditional

[1]Signature unforgeability can be implemented via public-key encryption techniques such as RSA [18].

one given for the crash-stop model [13], except integrity, which is adapted to the Byzantine case.[2]

2.3.2 Underlying Abstractions

We present here two abstractions on which we build our total order broadcast algorithm. The first abstraction is a traditional reliable broadcast primitive, i.e., a communication primitive that ensures all the properties of total order broadcast except the ordering property. The second abstraction is a new one called *weak interactive consistency* (WIConsistency for short). This abstraction encapsulates the agreement part of our total order broadcast algorithm. We simply give the specification of this abstraction here. Its algorithm is given in the next section.

Reliable Multicast Reliable broadcast is a communication primitive that allows a process to send messages to all processes, with the guarantee that all correct processes eventually deliver the same set of messages. Formally, reliable broadcast is defined through two primitives: *R_broadcast* and *R_deliver*. These primitives are the means to send and deliver messages according to the validity, integrity, and agreement properties of total order broadcast, when replacing A_broadcast and A_deliver with R_broadcast and R_deliver respectively. Such a primitive can be implemented with a simple rediffusion mechanism. That is, when some correct process p receives a correctly signed message m for the first time, p delivers and relays m to all processes.

Weak Interactive Consistency The second abstraction that we build on for our total order broadcast algorithm is a new abstraction that we call *weak interactive consistency (WIConsistency)*. This is a slight variation of the original *interactive consistency* abstraction [9]. Informally, in WIConsistency, each correct process first proposes its initial value to all the other processes. Then, every correct process eventually decides on the *same set of initial values*, which contains at least *one* initial value of some correct process.[3] Formally, WIConsistency is defined through a *propose* primitive. A correct process p invokes the *propose* primitive with its initial value as parameter to launch an instance of WIConsistency. When this instance is completed for some correct process p, we say that *p decide*s

[2]The original integrity property states the following: for any message m, every process A_delivers m at most once and only if the sender of m has indeed A_broadcast m. It is important to notice that the second clause of this property is not restricted to correct processes, i.e., the sender of m could either be correct or faulty. This is reasonable in the crash model, where faulty processes can only fail by halting. In the Byzantine model, however, even if a faulty process p invokes A_broadcast m, the external behavior of this invocation might look like an invocation of A_broadcast m' to some or all processes [13]. This can happen, for instance, if rather than sending the same message m to all processes, p sends different messages to different processes. More generally, because p can arbitrarily deviate from its specification, it is impossible to specify the actions p will perform as a faulty process. Consequently, to adapt the specification to the Byzantine model, the second part of the integrity property is restricted to actions performed by correct processes.

[3]In the original variant of interactive consistency, the set needs to contain the values of all correct processes [9].

and the decision value is the value returned by the *propose* primitive: this value is actually a set of initial values. WIConsistency satisfies the following properties:

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

2.3.3 Composing the Total Order Broadcast Algorithm

We show here how reliable broadcast and WIConsistency can be composed to obtain an algorithm that satisfies the properties of total order broadcast, in the presence of Byzantine failures. Interestingly, and thanks to our underlying abstractions, the principle of this composition follows the structure of the one proposed in Ref. [5], which, in a crash model, transforms consensus and reliable broadcast into total order broadcast.

2.3.3.1 The A_Broadcast Primitive (Algorithm 1) A correct process simply invokes *R_broadcast*(*m*) in order to A_broadcast a message *m*. That is, message *m* is first reliably broadcast to all processes.

Algorithm 1 *Total Order Broadcast Protocol: A_broadcast Primitive*

```
To execute A_broadcast (m):
R_broacast(m)
```

2.3.3.2 The A_Deliver Primitive (Algorithm 2) This primitive is executed by every process to A_deliver a message. Here, we use the reliable broadcast and the WIConsistency abstractions. Thanks to the reliable broadcast primitive, if one correct process delivers some message *m*, then all correct processes eventually *R_deliver m*. However, no order property is ensured by this primitive. For this reason, correct processes proceed in a sequence of stages, and they launch a new instance of WIConsistency at each stage. Intuitively, each stage *k* is responsible for A_delivering some of the R_delivered messages. Thus, every correct process launches the *k*th instance of WIConsistency with its local batch of reliably, yet not *A_delivered*, messages as proposed value. Eventually, every correct process decides on a set of proposed values, i.e., a set of batches.

Algorithm 2 *Total Order Broadcast Protocol: A_deliver Primitive*

```
1: Initialization
2: $R\_delivered \leftarrow \emptyset$
3: $A\_delivered \leftarrow \emptyset$
4: $k \leftarrow 0$
```

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

The algorithm relies on two concurrent tasks: *Task* 1 and *Task* 2. These tasks manipulate three sets of messages:

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

In *Task* 2 (lines 8–17), when a correct process *p* notices that the set *A_undelivered* is not empty, *p* launches a new instance of WIConsistency with *A_undelivered* as its initial value, and *k* as a sequence number that disambiguates concurrent executions of WIConsistency (line 11). Process *p* waits for the decision value (line 12), then collects the union of all messages that have been decided. This results in a list of messages noted *MsgSet* (line 13).

Process *p* then removes *incorrectly signed* messages, *mutant* messages and *redundant* messages from the set *MsgSet*. Before these messages are defined and the reasons for which they are removed are explained, it is important to point out that such messages can be safely removed because they cannot be sent by correct processes.

- **Incorrectly signed messages**. Removing all incorrectly signed messages avoids to A_deliver messages of which senders cannot be authenticated (line 14).
- **Mutant messages**. We also remove from *MsgSet* so-called mutant messages,[4] i.e., messages that have the same sequence number (identifier) but different

[4]The terminology of mutant messages was introduced by Kihlstrom et al. [14].

contents (line 15). Removing such messages contributes to ensuring the at-most-once semantics of total order broadcast. To achieve this, each message m that belongs to *MsgSet* is compared with the messages that have already been A_delivered and with other messages in *MsgSet*. If m is mutant with respect to some message that was already A_delivered, then m is removed from *MsgSet*. If m is mutant with respect to one or more messages in *MsgSet*, then m as well as *all its mutant messages* are removed from *Msgset*.

- **Redundant messages**. Nothing prevents Byzantine processes to repropose messages that have already been A_delivered; such messages are called redundant messages. Thus, any message m that belongs to both *A_delivered* and *MsgSet* is removed from *MsgSet* (line 16), to prevent the multiple A_delivery of m.

The removal of the above misleading messages results in the set $A\_deliver^k$: all messages that remain in $A\_deliver^k$ are then A_delivered according to some deterministic order (line 17). Finally, p updates its *A_delivered* set by augmenting it with all the messages in $A\_deliver^k$ (line 18).

2.4 WEAK INTERACTIVE CONSISTENCY

This section describes our weak interactive consistency (WIConsistency) algorithm. We first give an overview of the algorithm, then we present the abstractions and mechanisms underlying it, and finally we describe it in details.

2.4.1 Overview

Our WIConsistency algorithm has a preliminary step that precedes a rotating coordinator scheme. In the preliminary step, every correct process p_i sends its initial value through a message (p_i, v_i) to all processes. Then, p_i waits to collect $f + 1$ messages (p_j, v_j) from different processes (including itself). At the end of this waiting period, every process p_i has built a set, noted set_i, of $f + 1$ initial values. This set_i constitutes the estimate with which p_i starts the rotating coordinator scheme to reach a decision. The decided value is one of the estimates, i.e., a set of initial values. In the presence of at most f Byzantine processes, any decided value contains at least one initial value proposed by some correct process.

The rotating coordinator scheme proceeds in asynchronous rounds, each one being divided into two phases. Roughly speaking, Phase 1 is used to decide on a value proposed by some coordinator, whereas Phase 2 is triggered if Phase 1 has failed and is used to prepare for the new round. In Phase 1 of every round r, the correct processes try to decide on the estimate of the coordinator q of round r. The coordinator q starts by broadcasting its current $estimate_q$ of the decision. When a process receives $estimate_q$, it reissues (broadcasts) this value to all. Once a process has received $estimate_q$ from $N - f$ processes, it broadcasts a *decision* message containing $estimate_q$ and decides on it.

Phase 2 ensures that, if any process decides on $estimate_q$ in round r, then all correct processes that start round $r+1$ set their current estimate to $estimate_q$. This is ensured as follows. When a process "stops trusting" coordinator q, 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 whether one of the received estimates is the estimate of q.[5] If an estimate sent by q is found, the process adopts it and moves to round $r+1$.

2.4.2 Underlying Abstractions

Our WIConsistency algorithm relies on the following components: (1) a muteness failure detector, denoted by $\diamond M_{\mathcal{A}}$, to cope with crash-like behaviors (*muteness* behaviors), (2) an *Echo Broadcast* mechanism to cope with conflicting estimates and invalid estimates, (3) a *certification* mechanism to cope with invalid messages, and (4) a *filtering* mechanism to detect missing messages.

2.4.2.1 Muteness Failure Detector In a Byzantine environment, a process p might not receive an expected message from a process q, either because q has crashed, or because q decides not to send the message (i.e., q is not crashed but Byzantine). We encapsulate the ability of p to detect such a *muteness* behavior within a module, denoted by $\diamond M_{\mathcal{A}}$ and called a *muteness failure detector*. Here, we outline the specification of such a failure detector, whereas we devote Section 2.5 to a detailed discussion on this specification and its implementation.

Just like a crash failure detector is a distributed oracle that gives hints about crashed processes, a muteness failure detector can be viewed as a distributed oracle that gives hints about *mute* processes. Failure detector $\diamond M_{\mathcal{A}}$ outputs, at every correct process p, a set of processes that are considered to be *mute* with respect to p: when $\diamond M_{\mathcal{A}}$ outputs q at p, we say that *p suspects q*. Roughly speaking, $\diamond M_{\mathcal{A}}$ provides hints about Byzantine processes from which a correct process stops receiving algorithmic messages, with respect to some algorithm $\mathcal{A}$ ($\diamond M_{\mathcal{A}}$ might provide wrong hints). In our context, $\mathcal{A}$ represents the WIConsistency algorithm. We characterize $\diamond M_{\mathcal{A}}$ through the two following properties:

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

Unlike with crash failure detectors, the specification of muteness failure detectors depend on the algorithms using them. We come back to the inherent nature of this aspect in Section 2.6.

[5]An estimate message is made of the estimate and the identifier of the process proposing it.

2.4.2.2 Echo Broadcast At the beginning of each round, the current coordinator communicates its estimate to all processes (Phase 1). A Byzantine coordinator, rather than sending a unique estimate to all processes, may send them different estimates, which we call here *conflicting estimates*. The problem is to prevent correct processes from delivering, in the same round, conflicting estimates from a Byzantine coordinator.[6]

To cope with conflicting messages, we use an *Echo Broadcast* algorithm [17, 21]. In our context, the Echo Broadcast algorithm forces the coordinator of some round r to prove that it does not send conflicting messages to correct processes. More precisely, the coordinator of round r, say process q, initiates an Echo Broadcast by sending to all processes its estimate v in a signed message tagged *Initial*. Then, when a correct process p receives this *Initial* message for the first time, it extracts v and *echoes* it to q. That is, p sends to q a signed message tagged *Echo* that carries the received estimate v and the associated round number. Any additional *Initial* message received from q in this round is ignored by p. Once process q receives $N - f$ signed *Echo* messages for v, process q sends a signed message tagged *Ready* that carries the set of received signed *Echo* messages. No correct process p delivers v until it receives the *Ready* message that carries the $N - f$ expected *Echo*. This algorithm preserves correct processes from delivering conflicting estimates in the same round. The Echo Broadcast constitutes a means to preserve the agreement property *within a round.* Figure 2.1 illustrates the communication pattern of this algorithm in a given round r.

When some correct process, say p, receives an *Initial* message that carries an estimate that is not a set of $f + 1$ signed values (p_i, v_i), then p does not send an *Echo* message to the current coordinator. This prevents a Byzantine coordinator from successfully completing an Echo Broadcast on an *invalid* estimate, i.e., on an estimate with a $set_i = \{(p_j, v_j)_{p_j \in \pi}\}$ and $|set_i| = f + 1$.

2.4.2.3 Certification We introduce the notion of *certificate* to cope with *invalid* messages. Roughly speaking, an invalid message is one that is sent out of context. Before defining this notion more precisely, we first introduce the $\dot{\prec}$ relationship

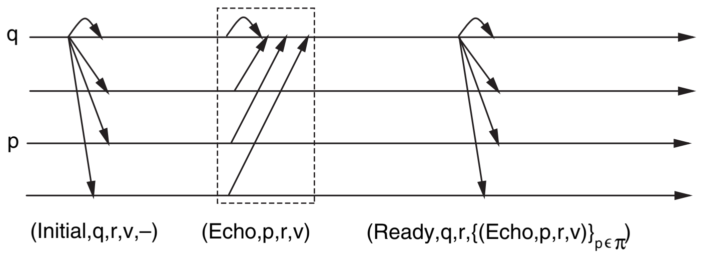

Figure 2.1 Exchange of messages in an echo broadcast.

[6]This is an instance of the well-known *Byzantine agreement* problem [2, 16].

between actions. Let $e1$ be the action of receiving a set of messages $sm1$ by some process p, and let $e2$ be the action of sending a message $m2$ by the same process p. We say that $e1$ *precedes* $e2$, noted $e1 \dot{\prec} e2$, if the action $e2$ of sending $m2$ is conditioned by the action $e1$ of receiving set $sm1$. From an algorithmic viewpoint, this definition can be translated as follows: "if receive $sm1$ then send $m2$." This definition is illustrated in Fig. 2.2.

In a trusted model, i.e., one that excludes malicious behaviors (e.g., the crash model), when some process performs an action $e2$, such that $e1 \dot{\prec} e2$, it is trivially ensured that (1) $e1$ happened before $e2$ and (2) $sm1$ was correctly taken into account to compute $m2$. In contrast, this is no longer guaranteed in a Byzantine model. A Byzantine process may perform $e2$ either without hearing about $e1$ or without taking into account the occurrence of $e1$. The resulting message $m2$ sent by such a Byzantine process is referred to as an invalid message. Coming back to our WIConsistency algorithm, if such invalid messages are not detected, the correctness of the algorithm may be compromised.

A *certificate* aims at proving the validity of any message $m2$ such that $sm1 \dot{\prec} m2$. The structure of the certificate appended to message $m2$ is therefore the set of all messages that compose $sm1$. Thus, when some correct process p receives message $m2$ and its associated certificate, knowing how $sm1$ should be taken into account to generate $m2$ allows p to verify whether $m2$ is a valid message. In the particular case of round $r = 1$, some messages $m2$ can be sent, although there was no previous delivery of $sm1$. In this case, such a message $m2$ is validated by an empty certificate.

At this point, a legitimate question is the following:is it possible for a Byzantine process to validate some message $m2$ with a false certificate? The answer is no, because no process (correct or not) can construct a certificate without the participation of a majority of correct processes in some round r. Indeed, any certificate used in the context of our Byzantine-resilient WIConsistency algorithm should satisfy two properties. First, all messages that compose a certificate are signed and sent in the same round r (the round is contained in each message). Second, every certificate should be composed of $N - f$ messages. Hence, having $f < N/3$, every certificate is composed of a majority of messages sent by correct processes. These properties ensure that the certificates constitute a way to preserve the agreement property of WIConsistency *across the rounds*.

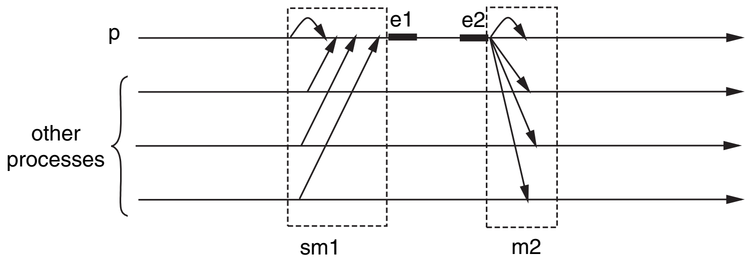

Figure 2.2 The $\dot{\prec}$ relationship.

2.4.2.4 Filtering *Missing* messages are messages that were voluntarily skipped by some Byzantine process, while this process is still sending other messages. For instance, a Byzantine coordinator may decide to skip sending the expected estimate message, while still sending other messages, e.g., replaying messages from previous rounds. Such behaviors can prevent the progress of correct processes.

The filtering of missing messages is based on the FIFO property of our reliable communication layer and on the following properties of the Byzantine-resilient WIConsistency 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 in round r always includes the value of r.

To illustrate how these two properties are used to detect missing messages, consider 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.

2.4.3 The WIConsistency Algorithm

As we explained, the algorithm (Algorithm 3) starts with a preliminary step during which every correct process collects a set of $f + 1$ signed initial values (p, v_p) from different processes. More precisely (lines 3-6), every correct process p sends all processes its initial value in a signed message (p, v_p). Then, p collects $f + 1$ correctly signed values from different processes and puts them into the set_p. This set_p is the valid estimate with which p starts participating in $Task1$.

After this step, the processes try to decide on a value through $Task1$ and propagate that decision through $Task2$. We describe these tasks below, first without depicting how certificates validate certain messages of the algorithm, and then we solely focus on the certification aspect. Hereafter, p designates any correct process executing Algorithm 3, whereas q designates any other process (correct or not) with which p interacts.

Task 1 (lines 10–43) Task 1 is divided into two phases. In Phase 1, every correct process tries to decide on the estimate of the current coordinator c_p (see Fig. 2.3). Throughout Task 1, a local predicate $Byzantine_p(q)$ is associated by p to every process q; initially, this predicate is *false*. As soon as p detects some misbehavior exhibited by q, like sending invalid messages or estimates, p sets its local predicate $Byzantine_p(q)$ to *true*. If p is informed that $Byzantine(c_p)$ is *true* or c_p is suspected at $2f + 1$ processes, then p proceeds to Phase 2 before moving to the next round (see Fig. 2.4).

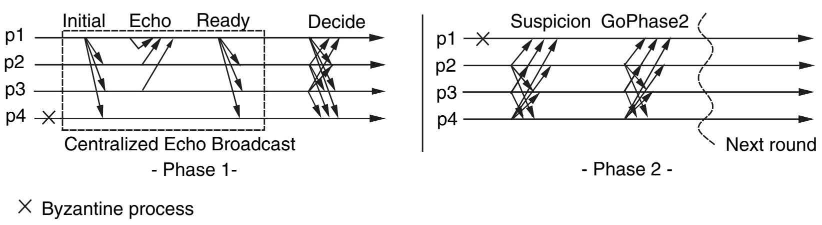

Figure 2.3 Exchange of messages in Phase 1.

2.4.3.1 Phase 1 (Lines 10–27) During Phase 1, if the current coordinator c_p is correct, it uses a centralized Echo Broadcast [17, 21] to send its estimate to all processes. First c_p sends its estimate set_{c_p} in a signed *Initial* message to all processes (line 11). When process p receives this message for the first time, p checks if set_{c_p} is a valid estimate carried by a valid *Initial* message, and if so, p sends an *Echo* message to c_p (lines 13-15). Then, once c_p collected $2f + 1$ *Echo* messages about set_{c_p}, c_p sends a *Ready* message to all processes (lines 16–18). At this point the Echo Broadcast algorithm is completed. Note that, when c_p sends a valid *Ready* message, we say that c_p has *successfully echo certified* its estimate. When process p receives a valid *Ready* message for the first time, it adopts set_{c_p} and relays the *Ready* message to all processes (lines 19–24). Once p received $2f + 1$ *Ready* messages containing c_p's estimate, p sends a *decide* message to all processes and decides on set_{c_p} (lines 25–27). If c_p is Byzantine, and does not send the appropriate messages, then p eventually moves to Phase 2.

2.4.3.2 Phase 2 (Lines 28–43) Phase 2 ensures that if any process decides on some estimate *set* during Phase 1 of round r, then every correct process that starts round $r + 1$ starts with its estimate equal to *set*. To proceed to Phase 2, a correct process p must learn that $Byzantine(c_p)$ is *true* or c_p is suspected at $2f + 1$ processes (lines 28–30). Before moving to Phase 2, p sends the *GoPhase*2 message to all (lines 31–33). This message carries the last valid estimate seen by p, i.e., the last successfully echo certified estimate seen by p. When p receives a valid *GoPhase*2 message, and p is still in Phase 1, p proceeds to Phase 2. Then, p waits for the reception of $2f + 1$ valid *GoPhase2* messages (lines 35–40). Among the $2f + 1$ valid *GoPhase*2 messages received, process p looks for the last estimate that was

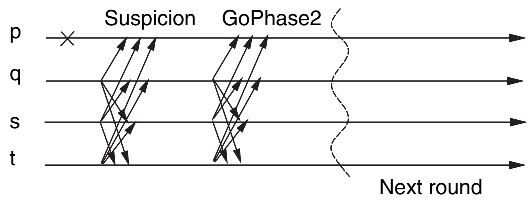

Figure 2.4 Exchange of messages in Phase 2.

successfully echo certified and, if such a value exists, p adopts it (line 41). Note that p might not find any successfully echo certified estimate, in which case p does not update its estimate. This is possible if, for instance, the coordinator is suspected before being able to echo certify its proposal. In that case, operation *Last_Successfully_EchoCertified(InitialCertif$_p$)* simply returns p's current estimate (set_p, $ReadyCertif_p$). Finally, p leaves Phase 2 and proceeds to the next round (lines 42–43).

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

2.4.4 About Certificates

In the following, we detail four kinds of messages in Algorithm 3 that require the use of certificates.

- **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 noted *InitialCertif*.

  For the first round ($r = 1$), since no rounds happened before, $Initial_r$ does not need any certificate to be considered as valid. 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 $r - 1$. Indeed, during this phase, process q waits for $2f + 1 GoPhase2$ messages, according to which it updates its estimate. We denote this set of $2f + 1$ $GoPhase2$ messages $Set\_GoPhase2_{r-1}$. Since $Initial_r$ must be sent after the reception of $Set\_GoPhase2_{r-1}$, we have that $Set\_GoPhase2_{r-1} \dot{\prec} Initial_r$. So, 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 $Set\_GoPhase2_{r-1}$ used for this update, each correct process can check the validity of the $Initial_r$ message received from q.

- **Ready message**. In round r, this message, noted here $Ready_r$, is sent by the current coordinator in the last step of the Echo Broadcast algorithm. This message is validated by a certificate named *ReadyCertif*.

  Before sending $Ready_r$, the current coordinator should wait for $2f + 1$ *similar Echo messages*, i.e., messages with identical estimates and identical round numbers. We note this set of $2f + 1$ *Echo* messages Set_Echo_r. Since $Ready_r$ must be sent after receiving Set_Echo_r, we have that $Set\_Echo_r \dot{\prec} Ready_r$. Consequently, the certificate *ReadyCertif* appended to $Ready_r$ is the collection of all messages that compose Set_Echo_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 $2f+1$ $Ready_{r'}$ messages from some round $r' \leq r$ before sending $Decide_r$. We note this set of $2f+1$ *Ready* messages $Set\_Ready_{r'}$. Since $Decide_r$ must be sent after the reception of $Set\_Ready_{r'}$, we have that $Set\_Ready_{r'} \dot{\prec} Decide_r$. Consequently, the certificate *DecideCertif* appended to $Decide_r$ is the collection of all messages that compose $Set\_Ready_{r'}$.

- **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 $2f+1$ *Suspicion* messages before sending $GoPhase2_r$. We note this set of $2f+1$*Suspicion* messages $Set\_Suspicion_r$. Since $GoPhase2_r$ must be sent after the reception of $Set\_Suspicion_r$, we have: $Set\_Suspicion_r \dot{\prec} GoPhase2_r$. Consequently, the certificate *GoPhase2Certif* appended to $GoPhase2_r$ is the collection of all messages that compose $Set\_Suspicion_r$.

Algorithm 3 Byzantine-Resilient WIConsistency Algorithm

```
1:  function propose($v_p$) {Every process p executes Task 1 and Task 2 concurrently}
2:  $InitialCertif_p \leftarrow \emptyset$
3:  $set_p \leftarrow \emptyset$                                          {Preliminary Phase}
4:  send $(p, v_p)$ to all
5:  wait until ($f$+1 processes $q$: received $(q,e_q)$)
6:      $set_p \leftarrow \{(q, e_q) \mid p$ received $(q, e_q)$ from $q\}$
7:  loop                                                               {Task 1}
8:      $CurrentRoundTerminated_p \leftarrow false$; $ReadyCertif_p \leftarrow \emptyset$;
        $GoPhase2Certif_p \leftarrow \emptyset$
9:      $DecideCertif_p \leftarrow \emptyset$; $coordSuspect_p \leftarrow false$; $c_p \leftarrow (r_p \bmod N) + 1$;
        $phase_p \leftarrow 1$
10:     if $c_p = p$ then
11:        send $(Initial, p, r_p, set_p, InitialCertif_p)$ to all
12:     while not $(CurrentRoundTerminated_p)$ do
13:        when receive Valid $(Initial, q, set_q, InitialCertif_q)$ from $q$
14:            if $(q = c_p) \wedge$ (no Echo message was sent in $r_p$ by $p$) then
15:               send $(Echo, p, r_p, set_q)$ to $c_p$
16:        when $(c_p = p) \wedge$ (for $2f$+1 distinct processes $q$: received $(Echo, q, r_p, set_p)$)
17:            $ReadyCertif_p \leftarrow \{(Echo, q, r_p, set_p) \mid p$ received $(Echo, q, r_p, set_p)$ from $q\}$
18:            send $(Ready, p, r_p, set_p, ReadyCertif_p)$ to all
19:        when receive Valid $(Ready, q, r_p, set_q, ReadyCertif_q)$
20:            $DecideCertif_p \leftarrow DecideCertif_p \cup \{(Ready, q, r_p, set_q, ReadyCertif_q)$
21:            if (first Ready message received) $\wedge$ $(p \neq c_p)$ then
22:               $ReadyCertif_p \leftarrow ReadyCertif_q$
23:               $set_p \leftarrow set_q$
24:               send $(Ready, p, r_p, set_p, ReadyCertif_p)$ to all
25:         else if $2f + 1$ Ready messages received from distinct processes then
26:            send ($Decide$, $p$, $r_p$, $set_p$, $DecideCertif_p$) to all
27:            $return(set_p)$
28:      when ($c_p \in \Diamond \mathcal{MA} \vee Byzantine_p(c_p)$) $\wedge$ ($not\ coordSuspected_p$)
29:         send ($Supicion$, $p$, $r_p$) to all
30:         $coordSuspected_p \leftarrow true$
31:      when ($phase_p = 1$) $\wedge$ for $2f + 1$ distinct processes $q$:
               received ($Supicion$, $q$, $r_p$))
32:         $GoPhase2Certif_p \leftarrow \{(Supicion, q, r_p) \mid p\ received\ (Supicion, q, r_p)$
                                  $from\ q\}$
33:         send (($GoPhase2$, $p$, $r_p$, $set_p$, $ReadyCertif_p$), ($GoPhase2Certif_p$)) to all
34:         $phase_p \leftarrow 2$; $InitialCertif_p \leftarrow \emptyset$
35:      when receive Valid (($GoPhase2$, $q$, $r_p$, $set_q$, $ReadyCertif_q$),
               ($GoPhase2Certif_q$))
36:         if $phase_p = 1$ then
37:            $phase_p \leftarrow 2$; $InitialCertif_p \leftarrow \emptyset$
38:            send (($GoPhase2$, $p$, $r_p$, $set_p$, $ReadyCertif_p$), ($GoPhase2Certif_q$))
                  to all
39:         $InitialCertif_p \leftarrow InitialCertif_p \cup (GoPhase2, q, r_p, set_q, ReadyCertif_q)$
40:         if $2f + 1$ GoPhase2 messages received from distinct processes then
41:            ($set_p$, $ReadyCertif_p$) $\leftarrow$ $Last\_Successfully\_EchoCertified(InitialCertif_p)$
42:            $currentRoundTerminated_p \leftarrow true$
43:            $r_p \leftarrow r_p + 1$
44:  when receive Valid ($Decide$, $q$, $r$, $set$, $DecideCertif_q$)                {Task2}
45:     send ($Decide$, $q$, $r$, $set$, $DecideCertif_q$) to all
46:     $return(set)$
```

2.5 MUTENESS FAILURE DETECTOR

A key abstraction underlying our weak interactive consistency algorithm is the muteness failure detector. Hereafter, we define the muteness failure model, then we proceed with a more precise specification of our muteness failure detector module, and then we discuss its implementation under partial synchrony assumptions.

2.5.1 The Muteness Failure Model

Intuitively, muteness characterizes faulty processes from which correct processes stop receiving algorithmic messages. As conveyed by Fig. 2.5, muteness failures represent a special case of Byzantine failures, yet a more general case than crash failures. A muteness failure may occur when a process simply crashes, or arbitrarily

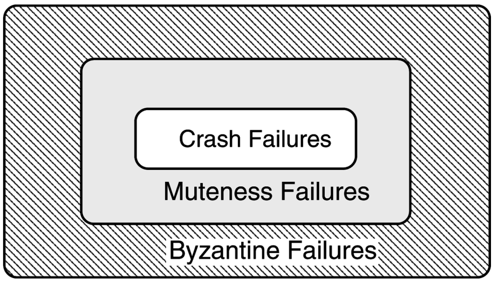

Figure 2.5 Inclusion relation between different types of failures.

decides to stop sending a specific subset of algorithmic messages to some or all correct processes, while still sending other messages.[7]

We now more precisely define the muteness behavior of a Byzantine process. Without loss of generality, we assume that the set of messages that can be generated by any algorithm $\mathcal{A}$ to have a specific syntax. Thereby, 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 $\mathcal{A}$. Although some messages sent by a Byzantine process may carry a semantic fault, we still consider them as $\mathcal{A}$ messages as long as they respect the syntax of messages that could be generated by algorithm $\mathcal{A}$.[8] Based on this definition, we define the notion of *muteness* as follows.

Mute Process A process q is mute to some correct process p, with respect to some algorithm $\mathcal{A}$, if there is a time after which p stops receiving $\mathcal{A}$ messages from q.

When some process q becomes mute with respect to some correct process p, we say that q fails by *quitting algorithm $\mathcal{A}$ with respect to process p.*

2.5.2 Muteness Failure Detector Specification

To simplify the presentation, we assume the existence of a real-time global clock outside the system: this clock measures time in discrete numbered ticks, whose

[7]A subtle cause of a muteness failure can be for instance the following: a Byzantine process sends algorithmic messages to a correct process, but its end of the communication channel skips some sequence number used to ensure FIFO ordering of messages. In this case, the FIFO channel on the other (correct) end no longer delivers the received messages to the correct process. Consequently, the correct process stops receiving messages from this particular Byzantine process. Retransmitting messages might in this case be useless because a Byzantine process can decide to systematically drop the same messages. At this stage, one can notice that there are various causes that can induce muteness failures. However, since we model failures and not faults, we merely ignore, in our specification, the causes of muteness and we only focus on the observable behavior of mute processes (from the point of view of a correct process).

[8]The conflicting estimates, defined in the previous section, are instances of messages that have a correct syntax but carry a semantic fault.

range is the set of natural numbers $\mathbb{N}$. This is merely a fictional device inaccessible to the processes of the set $\Pi = \{p_1, p_2, \ldots, p_N\}$.

2.5.2.1 Muteness Failure Pattern

A muteness failure pattern $F_{\mathcal{A}}$ is a function from $\Pi \times \mathbb{N}$ to 2^{Π}, where $F_{\mathcal{A}}(p, t)$ is the set of processes that quit algorithm $\mathcal{A}$ with respect to correct process p by time t. By definition, we have $F_{\mathcal{A}}(p, t) \subseteq F_{\mathcal{A}}(p, t+1)$. We also define $quit_p(F_{\mathcal{A}}) = \bigcup_{t \in \mathbb{N}} F_{\mathcal{A}}(p, t)$. We say that q is *mute* to p with respect to $\mathcal{A}$ if $q \in quit_p(F_{\mathcal{A}})$. We denote by $correct(F_{\mathcal{A}})$ the set of correct processes in failure pattern $F_{\mathcal{A}}$.

2.5.2.2 Muteness Failure Detector History

We define a *muteness failure detector history* as a function $H_{\mathcal{A}}$ from $\Pi \times \mathbb{N}$ to 2^{Π}, where $H_{\mathcal{A}}(p, t)$ denotes the set of processes suspected by a correct process p to be mute at time t.

2.5.2.3 Muteness Failure Detector

A *muteness failure detector* is a function $\mathcal{D}_{\mathcal{A}}$ that maps each failure pattern $F_{\mathcal{A}}$ to a set of muteness failure detectors histories. We define the class $\diamond M_{\mathcal{A}}$ of muteness failure detectors, such that any $\mathcal{D}_{\mathcal{A}} \in \diamond M_{\mathcal{A}}$ features the *mute $\mathcal{A}$-completeness* and the *eventual weak $\mathcal{A}$-accuracy* properties given below.

Mute $\mathcal{A}$-Completeness There is a time after which every process that is mute to any correct process p, with respect to $\mathcal{A}$, is suspected to be mute by p forever. This property is formally expressed as follows:

$$\forall F_{\mathcal{A}}, \forall H_{\mathcal{A}} \in \mathcal{D}_{\mathcal{A}}, \exists t \in \mathbb{N}, \forall p \in correct(F_{\mathcal{A}}) \wedge \forall q \in quit_p(F_{\mathcal{A}}),$$
$$\forall t' \geq t : q \in H_{\mathcal{A}}(p, t').$$

Eventual Weak $\mathcal{A}$-Accuracy There is a time after which some correct process p is no more suspected to be mute, with respect to $\mathcal{A}$, by any other correct process. This property is formally expressed as follows:

$$\forall F_{\mathcal{A}}, \forall H_{\mathcal{A}} \in \mathcal{D}_{\mathcal{A}}, \exists t \in \mathbb{N}, \exists p \in correct(F_{\mathcal{A}}),$$
$$\forall t' \geq t, \forall q \in correct(F_{\mathcal{A}}) : p \notin H_{\mathcal{A}}(q, t').$$

2.5.3 Muteness Failure Detector Implementation

In the layered architecture of Chandra and Toueg [5], the crash failure detector module is a black box (Figure ??). It can be implemented in a partial synchrony model in which we assume that there is a time after which process relative speeds and communication delays are bounded. The implementation is independent from the (consensus) algorithm using the failure detector.

In the following, we describe an implementation of our muteness failure detector module $\diamond M_{\mathcal{A}}$ under partial synchrony assumptions of Ref. [5]. Unlike in Ref. [5], however, our $\diamond M_{\mathcal{A}}$ implementation is not encapsulated within a black box: it depends on the algorithm using the module, i.e., A (e.g., on our WIConsistency algorithm). This is not surprising, given that the very specification of $\Diamond M_{\mathcal{A}}$ depends on the algorithm using it. We discuss below an implementation of $\Diamond M_{\mathcal{A}}$ that is generic in the sense that it can be used with a class of (*regular round-based*) algorithms, of which our WIConsistency algorithm belongs.[9]

Regular Round-Based Algorithms We define here the class of regular round-based algorithms, named $\mathcal{C}_{\mathcal{A}}$, in the context of which we give our $\Diamond M_{\mathcal{A}}$ implementation. The class $\mathcal{C}_{\mathcal{A}}$ is characterized by three attributes that must be featured by any algorithm $\mathcal{A} \in \mathcal{C}_{\mathcal{A}}$. These attributes are given below.

- **Attribute (a)**. Each correct process p owns a variable $round_p$ the range of which is the set of natural numbers $\mathbb{N}$.

  As soon as $round_p = n$, we say that *process p reaches round n* and *n is a reached round*. Then, until $round_p = n + 1$, process p is said to be *in round n*. Given any round n, if all processes are expecting at least one message from some process p, we say that p is a *critical process* of round n and its awaited messages are said to be *critical messages*. With these definitions, Attributes (b) and (c) can be stated as follows:
- **Attribute (b)**. Each process p is critical every k rounds, i.e., p is critical when $round_p \ mod \ k = 0$.
- **Attribute (c)**. Provided that at least $N - f$ correct processes reach round n, if the critical process p of round n is correct, then when p is in n, p sends at least one message to all processes in n.

Intuitively, Attribute (a) states that the algorithms within class $\mathcal{C}_{\mathcal{A}}$ proceed in rounds. Attribute (b) and (c) express, in terms of rounds, the very fact that any process should be critical an infinite number of times and that, when the critical process is correct, it should communicate with all processes. Hence, no correct process that executes $\mathcal{A}$ can be mute. In Attribute (c), the communication of correct critical processes is conditioned by the presence of $N - f$ correct processes, where N is the total number of processes participating in $\mathcal{A}$ and f is the maximum number of Byzantine processes tolerated by $\mathcal{A}$. That is, the actual value of $N - f$ depends on algorithm $\mathcal{A} \in \mathcal{C}_{\mathcal{A}}$ and represents the minimum number of correct processes required by $\mathcal{A}$ to execute an infinite number of rounds. As already mentioned, any algorithm $\mathcal{A}$ that aims at solving an agreement problem requires that $f < N/3$, which is the case for our WIConsistency algorithm. Our WIConsistency algorithm features attributes (a), (b), (c), and clearly belongs to class $\mathcal{C}_{\mathcal{A}}$.

A Parameterized Implementation for $\Diamond M_{\mathcal{A}}$ Algorithm 4 given below is an implementation for $\Diamond M_{\mathcal{A}}$, named $\mathcal{I}_{\mathcal{D}}$, assuming that algorithm $\mathcal{A}$ that uses $\mathcal{I}_{\mathcal{D}}$ belongs to class $\mathcal{C}_{\mathcal{A}}$. Algorithm 4 relies on a timeout mechanism and is composed of three concurrent tasks. Variable Δ_p, for some process p, holds the current

[9]Clearly, unless we restrict the class of algorithms that use $\Diamond M_{\mathcal{A}}$, the latter cannot be implemented even in a perfectly synchronous model. We shall come back to the rationale behind this in Section 2.6.

timeout and is initialized with some arbitrary value, $init_\Delta > 0$, that is the same for all processes. Furthermore, $\mathcal{I}_\mathcal{D}$ maintains a set $output_p$ of currently suspected processes and a set $critical_p$ containing the processes that p monitors (and hence adds to its $output_p$ set in case of suspicion). 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 from Δ_p ticks for some process q that is in $critical_p$, then q is suspected to be mute by p and is inserted in the $output_p$ set.

Correctness Figure 2.6 illustrates the interactions between algorithm $\mathcal{A}_p$, which designates algorithm $\mathcal{A}$ when executed by correct process p, and $\mathcal{I}_\mathcal{D}$. Besides queries to $\mathcal{I}_\mathcal{D}$ on suspected processes (arrow 1), our implementation $\mathcal{I}_\mathcal{D}$ handles two more interactions with $\mathcal{A}_p$, implemented by Task 2 and Task 3.

Algorithm 4 Implementation $\mathcal{I}_\mathcal{D}$ of Muteness Failure Detector $\diamond M_\mathcal{A}$

```
 1: {Every process p executes the following:}
 2: $\Delta_p \leftarrow init_\Delta$; $output_p \leftarrow \emptyset$; $critical_p \leftarrow \emptyset$;      {Initialization}
 3: for all $q \in critical_p$ do                                                      {Task 1}
 4:     if ($q \notin output_p$) ∧ (p did not receive "q-is-not-mute" during $\Delta_p$ ticks) then
 5:         $output_p \leftarrow output_p \cup q$
 6: when receive "q-is-not-mute" from $\mathcal{A}_p$                                  {Task 2}
 7:     if ($q \in output_p$) then
 8:         $output_p \leftarrow output_p - q$
 9: when receive $new\_critical_p$ from $\mathcal{A}_p$                                {Task 3}
10:     $critical_p \leftarrow new\_critical_p$
11:     $\Delta_p \leftarrow g_\mathcal{A}(\Delta_p)$
```

Each time p receives an $\mathcal{A}$ message from some process q (arrow 2), algorithm $\mathcal{A}_p$ delivers the message "q-is-not-mute" to $\mathcal{I}_\mathcal{D}$ (arrow 3). As a consequence, Task 2 removes process q from $output_p$ if 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. This operation is possible because algorithm $\mathcal{A}$ belongs to class $\mathcal{C}_\mathcal{A}$ and hence features Attributes (a), (b) and (c) introduced in Section 2.5.3.

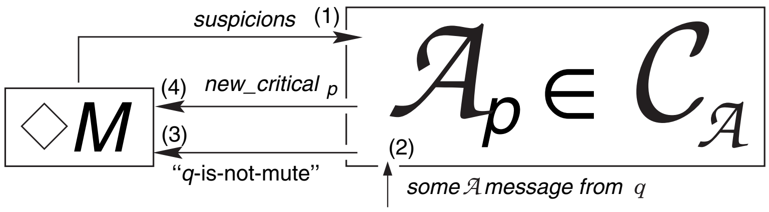
suspicions (1)
(4) new_critical p
(3)
"q-is-not-mute"
$\diamond M$
$\mathcal{A}_p \in \mathcal{C}_\mathcal{A}$
(2)
some $\mathcal{A}$ message from q

Figure 2.6 Interactions between $\mathcal{A}_p$ and $\diamond M_\mathcal{A}$.

2.5.3.1 Parameterization of $\mathcal{I}_\mathcal{D}$ Task 3 updates $critical_p$ with 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, $\Delta_\mathcal{A} : \mathcal{R} \to \mathcal{T}$, 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 $\Delta_\mathcal{A}(n) = 2^{n-1}\ init_\Delta$. Note that function $g_\mathcal{A}$ is precisely what parameterizes our implementation $\mathcal{I}_\mathcal{D}$ of muteness failure detector $\diamond M_\mathcal{A}$.

2.5.4 Interactions Between $\mathcal{A}_p$ and $\mathcal{I}_\mathcal{D}$

Proving the correctness of algorithm $\mathcal{I}_\mathcal{D}$ consists in proving that $\mathcal{I}_\mathcal{D}$ ensures the mute $\mathcal{A}$-completeness and the eventual weak $\mathcal{A}$-accuracy properties. The difficulty here lies in the very fact that, rather than approximating bounds on communication delays and process relative speeds, as does the implementation of [5] in the crash failure context, $\mathcal{I}_\mathcal{D}$ approximates the maximum delay between two consecutive $\mathcal{A}$ messages.

In some sense, $\mathcal{A}$ messages replace the traditional "*I-am-alive*" (heartbeat) messages in crash failure detector implementations. So, the interval between two consecutive $\mathcal{A}$ messages does not only depend on synchrony bounds, but also depends on the delay introduced by the semantics of algorithm $\mathcal{A}$ when sending two consecutive messages. This makes the proof of the correctness of $\mathcal{I}_\mathcal{D}$ rather tricky. A particularly difficult case is when the interval of time between two $\mathcal{A}$ messages also depends on the timeout value; e.g., whenever the timeout value increases, the interval between two $\mathcal{A}$ messages increases even further.

In the following, we first give a set of timing assumptions underlying our proof. In particular, these timing assumptions exclude the aforementioned case.

Assumptions for the Correctness Proof The correctness of $\mathcal{I}_\mathcal{D}$ relies on three main assumptions: (1) partial synchrony assumptions, (2) permanent participation assumptions, and (3) timing assumptions on algorithm $\mathcal{A}$ and on the timeout function of $\mathcal{I}_\mathcal{D}$, i.e., on function $\Delta_\mathcal{A}$. When our parameterized implementation $\mathcal{I}_\mathcal{D}$ is used with some algorithm $\mathcal{A}$, we have to prove that $\mathcal{A}$ and $\mathcal{I}_\mathcal{D}$, together, satisfy these timing assumptions. In particular, this implies that we have to determine an adequate function $g_\mathcal{A}$.

2.5.4.1 Partial Synchrony Assumptions We assume here the existence of a bound δ on communication delays and a bound ϕ on process relative speeds: these bounds are both unknown *and* hold only after some unknown time GST (global stabilization time) [7]. Such a weak partially synchronous system is also the one assumed by Chandra and Toueg for their implementability proof of $\diamond\mathcal{P}$ [5]. (Without such assumption, any non-trivial form of agreement is impossible, even with only one crash failure [8].) 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 to GST.

2.5.4.2 Permanent Participation We assume the permanent participation of $N-f$ correct processes. This is a theoretical assumption needed to prove liveness. 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 $\mathcal{A}$-accuracy *forever*. From a more practical perspective, we need the permanent participation property to hold long enough for the algorithm to terminate.[10] In fact, we only need $\mathcal{I}_{\mathcal{D}}$ to ensure $\Diamond M_{\mathcal{A}}$ properties *as long as no correct process terminates*. In the case of our WIConsistency algorithm for instance, once a correct process terminates, all correct processes are able to eventually terminate without using $\Diamond M_{\mathcal{A}}$ anymore.

2.5.4.3 Timing Assumptions on $\mathcal{A}$ and $\mathcal{I}_{\mathcal{D}}$ Here, we state three timing assumptions: the first and second ones are only related to algorithm $\mathcal{A}$, while the third one expresses a timing relation between algorithm $\mathcal{A}$ and $\mathcal{I}_{\mathcal{D}}$. Our WIConsistency algorithms satisfies these assumptions.

Before stating these assumptions, we first introduce the following definition: For any given round n, we say that n is *completed* for a correct critical process p of n, if all correct processes received all the critical messages expected from p in n.

Intuitively, Assumption (a) below states that, within any round n after GST, the critical messages of some correct critical process p are received by all correct processes within a fixed amount of time β, provided $N-f$ correct processes (including p) are in round n.

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

  Intuitively, Assumption (b) below states that, given some reached round n, the maximum time after GST needed for any correct process to reach n can be expressed by a function h.
- **Assumption (b)**. There exists a function $h: \mathcal{R} \rightarrow \mathcal{T}$ that maps any round reached by at least one correct process p, into the *maximum time* required by any correct process q in some round $m \leq n$ to reach n.

  Informally, Assumption (c) below states that, after GST, there exists a round n' after which the timeout $\Delta_{\mathcal{A}}(n)$ associated with any reached round $n \geq n'$ is larger and grows faster than $h(n)$.
- **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, and:

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

[10]Similarly, crash failure detector properties need only to hold *"long enough for the algorithm using them to achieve its goal"* [5, p. 228].

Corollary 2.5.1 From Assumptions (a), (b), and (c), we infer that:

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

In short, these timing assumptions avoid a detailed analysis of the internal structure of algorithm $\mathcal{A}$, in order to know when each $\mathcal{A}$ message is sent. We indirectly capture this information by quantifying the amount of time needed by some correct process to reach some round. Furthermore, to avoid false suspicions (after GST), we calibrate the timeout function. That is, a timeout function that eventually associates to each round a timeout value sufficiently large to allow all correct processes to reach this round.

Correctness Proof
Theorem 2.5.2 When used by any 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 timing Assumptions (a), (b) to (c), and assuming the permanent participation of $N-f$ correct processes.

PROOF We first prove the Mute $\mathcal{A}$-completeness property and then proceed with the Eventual weak $\mathcal{A}$-accuracy property.

2.5.4.4 Mute $\mathcal{A}$-Completeness By Attribute (b) 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 to the set of $critical_p$ of a correct process p. If q is mute to process p, it means p stops receiving $\mathcal{A}$ messages from q forever. Since the algorithm $\mathcal{A}_p$ of process p stops receiving $\mathcal{A}$ messages from q, then 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 q in its $output_p$ set (Task 1). Since p stops receiving $\mathcal{A}$ messages from q forever, then process q is never removed from $output_p$. Hence, process q is suspected forever to be mute by p. Therefore, there is a time after which the mute $\mathcal{A}$-completeness property holds forever.

2.5.4.5 Eventual Weak $\mathcal{A}$-Accuracy From mute $\mathcal{A}$-completeness and the assumption of the permanent participation of $N-f$ correct processes, we infer that 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 any two correct processes. From Algorithm 4, we know that q can only be added to $output_p$ in rounds where q is a critical process. From Corollary 2.5.1, 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 a critical process. Then, assume that p just reached n, i.e., p is at the beginning of round n. There are two possible cases:

1. **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 (including q) to reach round n, plus the time needed by round n to be completed for q.

As a 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 2.5.1, we infer that q will not be added to $output_p$ in n, nor in any future round where q will be critical.

2. **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. Thus, there exists a round $r < n$ such that (1) q is a critical process of r, and (2) p did not receive any critical messages sent by q in r (either q did not send them yet or p did not receive them yet). Since each correct process executes an infinite sequence of rounds, from the assumption that there are always $N - f$ correct processes participating in $\mathcal{A}$, and from Attribute (c), we know that process q eventually reaches round r as well as at least $N - f$ correct processes. Hence, q eventually sends a message to all in that round. So, there is a round $r' \geq n$, where p eventually receives q's messages sent in round r, 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 the above Case 1.

Therefore, a round $max(r', n)$ exists 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 $\mathcal{A}$-accuracy property holds forever. □

2.6 CONCLUDING REMARKS

The motivation of this work was to identify a set of abstractions to build upon a total order broadcast algorithm that tolerates *Byzantine* failures of processes and *asynchronous* periods of the network. We presented here such an algorithm based on two underlying fundamental abstractions: *weak interactive consistency* and *muteness failure detectors*. Roughly speaking, these abstractions are the counterpart of *consensus* and *crash failure detectors* in the modular approach of Ref. [5]. We evaluate here our layered architecture by discussing three aspects: (1) the modularity of our abstraction specifications, (2) the modularity of our abstraction implementations, and (3) the impact of our modularization on performance.

2.6.1 Modularity of Specifications

Whereas the specification of our weak interactive consistency abstraction is independent from the algorithm using it (i.e., total order broadcast), the specification of our muteness failure detector module is not. That is, the specification of our muteness failure detector module depends on the weak interactive consistency algorithm. We give below the rationale behind this dependency.

Unlike crash failure detectors [5], one cannot specify a Byzantine failure detector module independently from the algorithm using it. Indeed, a Byzantine 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). To illustrate this point, consider a process q that is part of a set Π of processes trying to agree on some value. Suppose now that q executes algorithm $\mathcal{A}$, which is proven to be correct, whereas all other processes in Π execute a different 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 r. The specification of Byzantine failures is intimately related to a specific algorithm and actually even to a specific process executing that algorithm. In short, Byzantine failures are not context-free, unlike crash failures. Consequently, a Byzantine failure detector specification cannot be, unlike a crash one, orthogonal to the algorithm using it.

One could imagine separating crash failures from other kinds of Byzantine failures and using crash failure detectors to track the former while leaving the latter to the algorithm itself. One could hope here to keep intact the modularity of the original model [5] and deal with failures that are of a more complex nature at a separate level. Unfortunately, this approach also does not make sense 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 [5]. 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, 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. Hence, there is no point in trying to track *crash* failures only.

Our approach is in a sense pragmatic because only part of the malicious behavior is captured by the failure detector, namely, *muteness* behavior. This approach enables us to capture a tricky part of malicious failures inside well-defined modules (muteness failure detectors), while restricting the dependency between the algorithm and the failure detector. Interestingly, and as we have shown, adequately defined, these muteness failure detectors encapsulate the amount of synchrony used to solve agreement problems like weak interactive consistency and indirectly total order broadcast.

2.6.2 Modularity of Implementations

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 still ensures its expected properties (e.g., completeness and accuracy). Because muteness failure detectors cannot be specified orthogonally to the algorithms using them, their implementation obviously depends on those algorithms. This dependency even restricts the set of algorithms that can make a meaningful use of failure detectors. Indeed, there are algorithms for which no implementation of our $\diamond M_{\mathcal{A}}$ muteness failure detector does make sense. These are for instance 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 $\mathcal{A}$'s specification, must eventually stop sending messages, i.e., must become mute with respect to all processes. Then the *Mute A-Completeness* property requires every correct process q to suspect p while the *Eventual Weak A-Accuracy* property requires that q eventually stops suspecting p. Clearly, the two properties become in contradiction making any implementation of $\diamond M_{\mathcal{A}}$ impossible.

What we suggest in this chapter can be viewed as a *gray-box* approach, with a parameterized implementation of a muteness failure detector $\diamond M_{\mathcal{A}}$ that is sufficient to solve weak interactive consistency and hence total order broadcast in a Byzantine environment. We require from the algorithms using our muteness failure detector $\diamond M_{\mathcal{A}}$ 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*. Note that even given those restrictions, we have shown 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. These assumptions, together with the regular round-based flavor, are featured by our weak interactive consistency algorithm.

It is also important to notice that, in Ref. [5], 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. Indeed, $\diamond M_{\mathcal{A}}$ can only cope with muteness failures, while leaving other malicious behaviors undetected, e.g., *conflicting messages.*, *invalid messages*, and *missing messages*. These had to be dealt with directly by the upper layer algorithms (i.e., weak interactive consistency and total order broadcast).

2.6.3 Impact on Performance

Many optimizations can easily be introduced in our total order broadcast algorithm to make it more efficient in steady state, i.e., when there is no suspicion and the primary (coordinator) does not change (we also typically talk about *nice* periods). This is the state of the system that is the most frequent in practice and for which algorithms are usually optimized. We give below two examples of optimizations.

1. Rather than deciding on a set of values, the processes can start by deciding only on one value, i.e., only on one batch of messages. That is, initially a traditional consensus is used rather than weak interactive consistency. If some process suspects that its batch is not being ordered, an instance of weak interactive consistency is then launched, i.e., as a fail-over mechanism.
2. Our Echo Broadcast algorithm can be decentralized to reduce the number of communication steps from three to two. Instead of going back to the sender after receiving its message and then having the sender send it to all, the processes would directly echo the message to all.

With these optimizations, three communication steps are needed to deliver a message (in a steady-state): just like in a decentralized 3PC [20]. In fact, the modularity of our approach helps identify the parts that could indeed be (safely) skipped in steady-state or (de)centralized. Furthermore, the very use of our weak interactive consistency abstraction (to agree on batches of messages) inherently enables us to gather messages and treat them together (in one shot), just like a group commit in transactional systems [10]. Through our agreement abstraction, we also circumvent the need for explicitly stabilizing messages before every new primary election after a fail-over (unlike in Ref. [3]). It would be interesting to explore the benefit of our modularisation in confining the use of public-key cryptography to fail-over states [3], or in recovering Byzantine processes [4].

REFERENCES

1. R. Baldoni, J.-M. Hélary, and M. Raynal. From crash fault-tolerance to arbitrary fault-tolerance: towards a modular approach. In *Symposium on Fault-Tolerant Computing Systems (FTCS)*, Los Alamitos, CA: IEEE Computer Society Press, 2000.
2. G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM*, 32(4): 824–840, October 1985.
3. M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, New Orleans, LA, February 1999.
4. M. Castro and B. Liskov. Proactive recovery in a byzantine fault tolerant system. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, San Diego, CA, October 2000.
5. T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2): 225–267, March 1996.
6. A. Doudou. *Abstractions for Byzantine-Resilient State Machine Replication.* PhD Thesis, 2000.
7. C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2): 288–323, April 1988.

8. M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32: 374–382, April 1985.

9. 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.

10. D. Gawlick and D. Kinkade. Varieties of concurrency control in IMS-VS fast path. *Database Engineering*, 8(2), 1985.

11. R. Guerraoui. Indulgent algorithms. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, July 2000.

12. R. Guerraoui and A. Schiper. Software-based replication for fault tolerance. *IEEE Computer*, 30(4): 68–74, April 1997.

13. V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broad-casts and related problems. Technical Report TR94-1425, Cornell University, Computer Science Department, May 1994.

14. 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*, vol. 3, pp. 317–326. IEEE, January 1998.

15. L. Lamport. The Part-time Parliament. Technical Report 49, Systems Research Center, Digital Equipement Corp., Palo Alto, September 1989.

16. L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3): 382–401, July 1982.

17. 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*, pp. 68–80, November 1994.

18. 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.

19. F. B. Schneider. Replication management using the state-machine approach. In *Distributed Systems* (edited by S. Mullender), ACM Press, pp. 169–197, 1993.

20. D. Skeen. Nonblocking commit protocols. In *International Conference on Management of Data (SIGMOD' 81)*, ACM Press, pp. 133–142, 1981.

21. S. Toueg. Randomized Byzantine Agreements. In *Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing*, pp. 163–178, August 1983.