Byzantine Failure Detection for Dynamic Distributed Systems

Murilo Santos de Lima, Fabiola Greve, Luciana Arantes, Pierre Sens

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Byzantine Failure Detection for Dynamic Distributed Systems

Murilo Santos de Lima — Fabíola Greve — Luciana Arantes — Pierre Sens

N° 7222

Mars 2010

Distributed Systems and Services

ISSN 0249-6399 ISRN INRIA/RR--7222--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
PARIS - ROCQUENCOURT

Byzantine Failure Detection for Dynamic Distributed Systems

Murilo Santos de Lima , Fabíola Greve * , Luciana Arantes , Pierre Sens †

Theme : Distributed Systems and Services
Networks, Systems and Services, Distributed Computing
Équipes-Projets Regal

Rapport de recherche n° 7222 — Mars 2010 — 21 pages

Abstract: Byzantine failure detectors provide an elegant abstraction for implementing Byzantine fault tolerance. However, as far as we know, there is no general solution for this problem in a dynamic distributed system over wireless networks with unknown membership. This paper presents thus a first Byzantine failure detector for this context. The protocol has the interesting feature to be time-free, that is, it does not rely on timers to detect omission failures. This characteristic favors its scalability and help to deal with the dynamics and unpredictability of those networks.

Key-words: failure detectors, Byzantine failures, dynamic distributed systems, wireless networks, self-organizing systems

* DCC - Computer Science Department / Federal University of Bahia
† LIP6 - University of Paris 6 - INRIA - CNRS

Centre de recherche INRIA Paris – Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex
Téléphone : +33 1 39 63 55 11 — Télécopie : +33 1 39 63 53 30

Détection des fautes byzantines pour les systèmes répartis dynamiques

Résumé : Les détecteurs de défaillances Byzantines offrent une abstraction élégante pour implanter la tolérance aux fautes Byzantines. Cependant, à notre connaissance, il nŠexiste pas de solution générale pour ce problème dans un système réparti dynamique. Cet article présente un premier détecteur de défaillance Byzantin pour ce type dŠenvironnement. Le protocole proposé est asynchrone dans le sens où les processus nŠutilisent pas de temporisateur pour détecter les fautes. Cette caractéristique rend le protocole extensible et adaptable.

Mots-clés : détecteurs de fautes, fautes Byzantines, systèmes répartis dynamiques, réseaux sans fil, systèmes auto-organisant

1 Introduction

Modern distributed systems, deployed over ad-hoc networks, such as wireless mesh networks (WMN), wireless sensor networks (WSN) are inherently dynamic. They are composed by a dynamic population of nodes, which randomly join and leave the network, at any moment of the execution, so that only a partial knowledge about the system's properties can be retained. Global assumptions, such as the knowledge about the whole membership, the maximum number of failures, complete or reliable communication, are no more realistic. Therefore, classical distributed protocols are no longer appropriate for this new context, since they make the assumption that the whole system is static and its composition is previously known.

Byzantine fault tolerance (BFT) [LSP82] plays an important role on the development of dependable dynamic distributed systems. It deals with a number of security problems by tolerating the presence of corrupted processes, which may behave in an arbitrary manner, trying to hinder the system to work accordingly to its specification. The implementation of BFT tecniques is a major challenge on dynamic distributed systems as many factors favor the action of malicious agents, e.g., the dynamic population, the wireless communication medium, the necessity to cooperate in order to achieve fundamental tasks (e.g., routing).

An *Unreliable failure detector*, namely FD, is a fundamental service that provides an elegant approach to design dependable and modular systems under dynamic environments [CT96]. It gives hints on which processes in the system are faulty and exempts the overlying protocol to deal with the failure treatment and synchrony requirements. But, differently from FDs for crash or "benign" failures, in which the FD implementation and practical assumptions can be addressed independently, FDs for Byzantine failures must rely the detection on the application algorithm that uses it as an underlying oracle. This is because the detection is made according with the message contents and communication pattern followed by the specific algorithm. This inherently symbiosis between the FD and application algorithm is perhaps at the cause of the very little work done until now in the domain of Byzantine fault detection.

Malkhi *et al.* [MR97] extend the theory of [CT96] and define a FD able to identify processes that prevent the progress of the algorithm using it. Doudou *et al.* [DS98] introduce the concept of *muteness failure detectors* in which the oracle detects when a process is mute, that is, when it ceases sending messages required by the algorithm. While these two works are restricted to a small subset of failures, the work of Kihlstrom *et al.* [KMMS03] extends the classical model of FDs for crash failures [CT96] to propose new classes able to consider more generic Byzantine failures and to solve consensus. Baldoni *et al.* [BHRT03] provides a framework to solve consensus which integrates muteness failure detectors (for mute crashes) and a byzantine behavior detector (for other Byzantine failures). All these works consider a classical distributed system in which the communication graph is complete and neither the number nor set of participants are known. A few exception is [AHNRR02], but it solves only a subset of the Byzantine failure detection problem to the specific application of routing.

Recently, Haeberlen *et al.* [HKD07] presented the PeerReview system and propose a concrete solution to the Byzantine fault problem based on the use of accountability to detect and expose node faults. The solution is suitable for dynamic systems which spam multiple administrative domains, e.g., P2P and

overlay multicast systems, but does not consider the case of systems in which the membership is unknown since the beginning. In a subsequent work [HK09], the same authors provide a formal study of the generic fault detection problem. They give a formal definition of commission (or security) and omission (or progress) faults [KMMS03] and identify some bounds on the costs of solving a weak definition of failure detection problem in asynchronous systems with authenticated channels.

All the Byzantine FDs proposed so far adopt the *timer-based* model to detect progress failures. This is a common design principle which supposes that eventually some bound on the transmission delays will permanently hold. However, these bounds are not known and they hold only after some unknown time [CT96]. An alternative approach suggested by [MMR03] is *time-free* and considers that the system satisfies a message exchange pattern on the execution of a communication primitive. It does not rely on timers to detect crash failures and assumes that the responses from some stable known process to a query launched by other processes permanently arrive among the first ones. This idea has been exploited by [SAB+08] to develop a FD for dynamic networks, but for the crash failure model. While the timer-based approach imposes a constraint on the physical time (to satisfy message transfer delays), the time-free approach imposes a constraint on the logical time (to satisfy a message delivery order). Both approaches (timer-based and time-free) are orthogonal and cannot be compared.

In dynamic networks, since the communication delays may frequently vary due to failures, arrivals and departures of nodes, the statement of the transmission bounds required by the timer-based detection becomes a big challenge. In this sense, the time-free model appears as a suitable alternative for being used in a dynamic set [MRT+05].

This paper advocates the use of the time-free approach to provide Byzantine failure detection. It proposes a model, a specification and an algorithm to implement an unreliable Byzantine FD adequate for dynamic networks with unknown membership and partial communication. To the best of our knowledge, the adoption of a time-free Byzantine detection in networks with unknown membership is novel and this paper provides a first insight towards the understanding and implementation of such an approach.

The rest of the paper provides the model (Section 2), time-free additional assumptions (Section 3), the algorithm (Section 4), its correctness proofs (Section 5) and conclusion (Section 6).

2 Model for Failure Detection in Dynamic Networks

We are particularly interested in systems deployed over wireless ad-hoc networks, such as WSNs and WMNs. The system is a set of nodes communicating by broadcasting messages via a packet radio network.

Finite arrival model [Agu04]. The network is a dynamic system composed of infinitely many processes; but each run consists of a finite set Π of $n > 3$ nodes, namely, $\Pi = \{p_1, \ldots, p_n\}$. This model properly express what does happen in dynamic networks since nodes join and leave the system as they wish.

The membership is unknown. Processes are not aware about Π or n, because, moreover, these values can vary from run to run [Agu04]. There is one process per node; each process knows its own identity, but it does not necessarily knows the identities of the others. Nonetheless, they can make use of the broadcast facility of the wireless medium to know one another. Thus, we consider that a process knows a subset of Π, composed with nodes with whom it previously communicated.

Processes are subject to *Byzantine failures* [LSP82], i.e., they can deviate arbitrarily from the algorithm they are specified to execute and work in collusion to corrupt the system behavior. A process that does not follow its algorithm specification is said to be *Byzantine*; otherwise, it is *correct*. In particular, a Byzantine process may send messages not previously defined by its algorithm or may omit to send messages it is supposed to. In this sense, a process that crashes can be regarded as Byzantine. Notice that Π is a partition of correct and Byzantine processes. Every process is uniquely identified and a Byzantine process cannot obtain more than one identifier. Thus, it is impossible to launch a *sybil attack* against the system [Dou02].

The system is asynchronous. There are no assumptions on the relative speed of processes or on message transfer delays. There is no global clock, but to simplify the presentation, we take the range $\mathcal{T}$ of the clock's tick to be the set of natural numbers.

Communication graph is dynamic. The network is represented by a communication graph $G = (V, E)$ in which $V = \Pi$ represents the set of nodes and E represents the set of logical links. The topology of G is dynamic due to arbitrary joins, leaves and failures. A link between nodes p_i and p_j is bidirectional, meaning that p_i is within the wireless transmission range of p_j and vice-versa. Let R_i be the transmission range of p_i, then all the nodes that are at distance at most R_i from p_i in the network are considered 1-hop *neighbors*, belonging to the same *neighborhood*. We denote N_i to be the set of 1-hop *neighbors* from p_i and $|N_i|$ its cardinality; thus, $(p_i, p_j) \in E$ *iff* $(p_i, p_j) \in N_i$.

Communication is fair-lossy. Local wireless channels are *authenticated* and *fair-lossy*. Thus, every process p_i holds a private key $\mathcal{K}_i$ with which it can sign its messages; and every process in the system can obtain the public key of every other node in order to authenticate the sender of any signed message [Sch96][1]. Moreover, a message m sent by a correct process p_i an infinite number of times is received by every correct process p_j in its neighborhood an infinite number of times, or p_j is Byzantine. In addition, there is no message duplication, modification or creation; this means that a Byzantine node is not allowable to interfere on message transmissions by correct processes, and even if it sends multiple versions of a message, the message will be perceived by the others as only one message with the same contents [Koo04, BV07]. The Fair-lossy assumption seems to be unrealistic for the dynamic environment; above all, wireless channels are inherently unreliable and can in addition suffer a number of attacks, e.g., a malicious node can raise a collision attack in messages sent by honest nodes, preventing reception. Notice however that some works about ensuring reliability under wireless channels have recently appeared. They advocate a "local" fault model, instead of a "global" fault model, as an adequate

[1] Without authenticated channels, it is not possible to tolerate process misbehavior in an asynchronous system since a single faulty process can play the roles of all other processes to some (victim) process.

strategy to deal with the dynamism and unreliability of wireless channels in spite of Byzantine failures [Koo04, PP05, BV05, BV07, BV10], defining bounds on the maximum number of local failures in order to reliably delivery data. Precisely, [BV05, BV10] shows that it is possible to achieve reliable broadcast if less than 1/4 of nodes in any neighborhood are Byzantine and impossible otherwise. Knowing that f_i is the maximum number of faulty processes in p_i's neighborhood, $f_i < |N_i|/4$. Locality of failures can be interpreted as an uniform distribution of failures across the network and represents more accurately the reality of wireless channels. This is the approach followed in our work.

2.1 Stability Requirements

One important aspect on the design of FDs for dynamic networks concerns the time period and conditions in which processes are connected to the system. During unstable periods, certain situations, as for example, connections for very short periods or numerous joins or leaves along the execution (characterizing a churn) could block the application and prevent any useful computation. Thus, to implement any global computation, the system should present some stability conditions that when satisfied for longtime enough will be sufficient to satisfy the requirements of the application and terminate.

In order to implement FDs with an unknown membership, processes should interact with some others to be known. If there is some process such that the rest of processes have no knowledge whatsoever of its identity, there is no algorithm that implements a FD with weak completeness [JAF06]. *Completeness* characterizes the FD capability of suspecting every faulty process permanently. In this sense, the characterization of the *actual membership* of the system, that is, the set of processes which might be considered for the computation is of utmost importance.

We consider then that a process p_i *joins* the network at some point $t \in \mathcal{T}$ in time. Subsequently, p_i must somehow communicate with the others in order to be known. In a wireless network, this can be done by simply broadcasting its identity to the neighbors. Due to this initial communication, every process p_j is able to gather an initial partial knowledge $\Pi_j \subseteq \Pi$ about the system's membership which increases over the time along p_j's execution. Afterwards, when p_i *leaves* the network at time $t' > t$, it can re-enter the system with a new identity, thus, it is considered as a new process. Processes may join and leave the system as they wish, but the number of re-entries is bounded, due to the finite arrival assumption.

Definition 1 *Membership. Let $t, t' \in \mathcal{T}$. Let $UP(t) \subseteq \Pi$ be the set of processes that are in the system at time t, that is, after have joined the system before t, they neither leave it nor crash before t. Let p_i, p_j be nodes. Let Π_j be p_j's partial knowledge about the system's membership. The membership of the system is the* KNOWN *set.*

$\text{BYZANTIN} \stackrel{def}{=} \{\forall p_i : p_i \text{ is Byzantine}\}$.

$\text{STABLE} \stackrel{def}{=} \{p_i : \exists t, t', s.t.\ \forall t' \geq t,\ p_i \in UP(t') \wedge p_i \notin \text{BYZANTIN}\}$.

$\text{FAULTY} \stackrel{def}{=} \{p_i : \exists t, t',\ t < t',\ (p_i \in UP(t) \wedge p_i \notin UP(t')) \vee p_i \in \text{BYZANTIN}\}$.

$\text{KNOWN} \stackrel{def}{=} \{p_i : (p_i \in \text{STABLE} \cup \text{FAULTY}) \wedge (p_i \in \Pi_j, p_j \in \text{STABLE})\}$.

The actual membership of the system is in fact defined by the KNOWN set. A process is *known* if, after have joined the system, it has been identified by some stable process. A *stable* process is thus a correct process that, after had entered the system for some point in time, never departs; otherwise, it is *faulty*. Following recent works about Byzantine radio communication [Koo04, BV10], we adopt a local fault model in which f_i is the maximum number of faulty processes in p_i's neighborhood.

Assumption 1 *Network with Byzantine coverage. Let* $G(\text{KNOWN} \cap \text{STABLE}) = G(S) \subseteq G$ *be the graph obtained from the stable known processes. Then, a system has Byzantine coverage if and only if* $\exists t \in \mathcal{T}$*, s.t: (1) there is a node-disjoint path between every pair of stable processes* $p_i, p_j \in G(S)$*; (2) the minimum degree of a node* p_i *in* G *is* $|N_i| > 2f_i$*.*

Connectivity assumption (1) states that, in spite of changes in the topology of G, from some point in time t, the set of known stables forms a *strongly connected component* in G. This is a frequent assumption, mandatory to ensure reliable dissemination of messages to all stable processes and thus to ensure the global properties of the failure detector [CT96, JAF06, MRT+05, Koo04, BV05, BV10].

Connectivity assumption (2) establishes a bound to tolerate Byzantine faults. It is a guarantee that information from/to process p_i is going to be sent/received to/from a minimum of stable nodes in its neighborhood. Precisely, at least f_i+1 stable nodes can communicate with p_i, ensuring that initially $p_i \in \Pi_j$ of at least $f_i + 1$ stable processes. Afterwards, if p_i is faulty, eventually at least $f_i + 1$ stable processes will suspect p_i and may spread the suspicion to the remaining of the system, so that the *completeness* property of the FD can be satisfied.

2.2 Byzantine Failures

Two requirements must be satisfied in a system prone to Byzantine failures: (i) correct processes must have a coherent view of the messages sent by every process; (ii) correct processes must be able to verify if a message is consistent with the requirements of the algorithm in execution. Thus, Byzantine failure detection is defined as a function of some algorithm $\mathcal{A}$. The first requirement may be addressed by two distinct techniques: information redundancy or unforgeable digital signatures; the second requirement can be met by adding certificates to the messages, so that its content may be validated [Sch96].

Two superclasses of Byzantine failures can be distinguished [KMMS03]: *detectable*, when the external behavior of a process provides evidence of the failure and *non-detectable*, otherwise. This work deals with detectable failures. They are classified in *omission* (or *progress*) failures and *commission* (or *security*) failures. Omission failures hampers the termination of the computation, since a faulty process does not send the messages required by the specification or sends it only to part of the system. Commission failures violate invariant properties to which processes must obey, and can be defined as the noncompliance of one of the following restrictions: (i) a process must send the same messages to every other; (ii) the messages sent must conform the algorithm $\mathcal{A}$ under execution.

2.3 Byzantine Unreliable Failure Detector Definition

Kihlstrom *et al.* [KMMS03] define Byzantine failure detector classes which differ from those described by Chandra and Toueg [CT96], since the latter deals only with crash failures. Let $\mathcal{A}$ be an algorithm that uses the failure detector as a underlying module. The class $\Diamond\mathcal{S}(Byz, \mathcal{A})$ is an adaptation of the $\Diamond\mathcal{S}$ class to Byzantine failures. It is the focus of our work. Nonetheless, its properties should be adapted to a dynamic network. With this aim, we define the class of *Eventually Strong Byzantine Failure Detectors with Unknown Membership*, namely $\Diamond\mathcal{S}^M(Byz, \mathcal{A})$. It keeps the same properties of $\Diamond\mathcal{S}(Byz, \mathcal{A})$, except that they are now valid to known processes. Informally, these properties are:
• *Strong Byzantine completeness* (for $\mathcal{A}$): eventually, every stable known process suspects permanently every process that has detectably deviated from $\mathcal{A}$;
• *Eventual weak accuracy*: eventually, one stable known process is never suspected by any stable known process.

Definition 2 *Eventually Strong Byzantine FD with Unknown Membership ($\Diamond\mathcal{S}^M(Byz, \mathcal{A})$). Let $t, t' \in \mathcal{T}$. Let p_i, p_j be nodes. Let $susp_j$ be the list of processes that p_j currently suspects of being faulty. The $\Diamond\mathcal{S}^M(Byz, \mathcal{A})$ class contains all the failure detectors that satisfy:*

Strong Byzantine completeness (for $\mathcal{A}$) $\stackrel{def}{=}$ $\{\exists t, t', s.t.\ \forall t' \geq t,\ \forall p_i \in \text{KNOWN} \cap \text{FAULTY} \Rightarrow p_i \in susp_j,\ \forall p_j \in \text{KNOWN} \cap \text{STABLE}\}$;

Eventual weak accuracy $\stackrel{def}{=}$ $\{\exists t, t', s.t.\ \forall t' \geq t,\ \exists p_i \in \text{KNOWN} \cap \text{STABLE} \Rightarrow p_i \notin susp_j,\ \forall p_j \in \text{KNOWN} \cap \text{STABLE}\}$.

3 Towards a Time-Free $\Diamond\mathcal{S}^M(Byz, \mathcal{A})$ Byzantine Failure Detector

3.1 Local Message Exchange Pattern

Most of the protocols for crash failure detection are based on the exchange of heartbeat messages by the failure detector. Nevertheless, in a Byzantine environment such a mechanism is no longer enough. A Byzantine process may correctly answer the failure detector messages, yet without guaranteeing progress and safety to the algorithm under execution. Therefore, the failure detection should be based on the pattern of the messages sent during the execution of the algorithm $\mathcal{A}$ which uses the failure detector as a building block [KMMS03].

We advocate the use of the time-free approach to raise suspicions and propose a FD protocol whose detection does not use timers but is based on the exchange of messages required by algorithm $\mathcal{A}$. Thus, when algorithm $\mathcal{A}$ requires the processes to exchange a message m, every process p_i waits until the reception of m from at least α_i distinct senders; for the remaining processes ($\in \Pi_i$), it raises an omission failure suspicion. In this case, p_i will send a SUSPICION message to processes in its neighborhood, carrying out its local view about suspicions. The detection follows a local message exchange pattern, i.e., between the nodes in the neighborhood [SAB+08]; thus, α_i corresponds to the minimum amount of stable known nodes in the neighborhood of p_i, i.e., $\alpha_i = N_i - f_i$. Knowing that $|N_i| > 2f_i,\ \alpha_i \geq f_i + 1$ (from the Byzantine coverage). The actual value of α_i

depends on the type of dynamic network considered (WSN, WMN) as well as the topology of the network during execution.

3.2 Characteristics of the Overlying Algorithm

It is important to notice that, since the detection follows an asynchronous pattern in which suspicions are based on the message exchange, the communication pattern followed by algorithm $\mathcal{A}$ must be distributed. That is, all nodes must exchange messages, following a $n \rightarrow n$ pattern. So, the protocol followed by $\mathcal{A}$ should be symmetrical. Since the proposed detector uses a local message exchange pattern, this symmetrical communication must occur at least between processes in the same neighborhood. Thus, we conjecture to be impossible to detect omission failures if such a pattern is of the form $1 \rightarrow n$. That is, if at any moment of the algorithm execution, only one process is required to send messages. Otherwise, one could not distinguish an omission failure from a delay on the delivering of the message from that process, since the underlying system is asynchronous [CT96]. Thus, we identify the following conjecture, and if it is correct, it derives the following corollary.

Conjecture 1 *In an asynchronous system, it is not possible to detect Byzantine omission failures time-freely if algorithm $\mathcal{A}$ allows a message exchange pattern $1 \rightarrow n$; that is, if algorithm $\mathcal{A}$ requires only a single process to send messages to the remaining.*

Corollary 1 *The time-free approach of Byzantine failure detection may only be adopted by symmetrical protocols.*

In practice, if the communication pattern followed by algorithm $\mathcal{A}$ is not distributed, one can simulate this pattern by requiring processes to relay the messages received to the other processes. Thus, when a process receives a message for the first time, before proceed with the computation, it must send it to all processes.

3.3 Behavioral System Property

With a time-free approach [MMR03], in order to satisfy *eventual weak accuracy* property of the $\Diamond\mathcal{S}^M(Byz, \mathcal{A})$ FD class, there must exists a stable process p_i whose messages from some point on are always among the first messages received by its neighbors, at every request of $\mathcal{A}$. Thus, eventually p_i will no longer be suspected by any stable process. The *Byzantine responsiveness* property characterizes this desired behavior.

Property 1 *$Byz\mathcal{RP}$ (Byzantine Responsiveness Property). Let $t'', t', t \in \mathcal{T}$. Let $rec\_from_j^{t'}$ ($rec\_from_j^{t''}$) be the set of processes from which p_j received the message required by $\mathcal{A}$ at its last step in execution until $t(t'')$. Let $p_i \in$ KNOWN $\cap$ STABLE. p_i satisfies $Byz\mathcal{RP}$ at time t if:*

$Byz\mathcal{RP}^t(p_i) \stackrel{def}{=} \forall t' \geq t, \forall t'' > t', p_i \in rec\_from_j^{t'} \Rightarrow p_i \in rec\_from_j^{t''} \vee$ *p_j is faulty.*

Thus, the following behavioral assumption should be satisfied in the network in order to implement $\Diamond\mathcal{S}^M(Byz, \mathcal{A})$: $\exists p_i \in \text{KNOWN} \cap \text{STABLE} : Byz\mathcal{RP}^t(p_i)$ eventually holds.

As a matter of comparison, in the timer-based model, the $Byz\mathcal{RP}^t(p_i)$ property would approximate the following: there is a time t after which the output channels from a stable process p_i to every other process p_j that knows p_i are eventually timely. That assumption coincides to the classical one used to implement $\Diamond S$ FDs in traditional networks [MR97, KMMS03, BHRT03].

3.4 Practical Issues

WSNs and WMNs are a good examples of networks who would satisfy the assumptions of our model, specially the $Byz\mathcal{RP}$ property and network assumptions. In a WMN, the client nodes move around a fixed set of nodes (the backbone of the network) and each mobile node eventually connects to a fix node. A WSN is composed of stationary nodes and can be organized in clusters, so that communication overhead can be reduced; one node in each cluster is designated the cluster head (CH) and the other nodes, cluster members (CMs). Communication inter-clusters is always routed through the respective CHs which act as gateway nodes and are responsible for maintaining the connectivity among neighboring CHs. For all these platforms, special nodes (the fixed nodes for WMN, CHs for WSN) eventually form a strongly connected component of stable nodes; additionally, some of these nodes can be regarded as fast, so that they will always answer messages faster than the other nodes, considered as slow nodes. Thus, one of these fast nodes may satisfy the $Byz\mathcal{RP}$ property. The stability conditions and the $Byz\mathcal{RP}$ may seem strong, but in practice they should just hold during the time the application needs the completeness and accuracy properties of FDs of class $\Diamond\mathcal{S}^M(Byz, \mathcal{A})$, as for instance, the time to execute a consensus algorithm [KMMS03].

4 A Time-Free Byzantine Failure Detector of the Class $\Diamond\mathcal{S}^M(Byz, \mathcal{A})$

4.1 Design Principles

Suspicion Generation Every SUSPICION message of an omission faulty raised over p_j is related to a message m required by $\mathcal{A}$. That is, p_j is suspected of not sending the messages of $\mathcal{A}$ it should. Thus, messages must have unique identifiers. Suspicions are propagated on the network and a stable process will adopt a suspicion not generated by itself if and only if it receives it properly signed from at least $f_i + 1$ different senders. This requirement denies a Byzantine process to impose suspicions on stable processes. Since the network has a Byzantine coverage, at least $(f_i + 1)$ neighbors of p_i are stable and shall spread a suspicion of its failure to their respective neighbors. Since moreover there is a path formed only by stable processes between any stable process, eventually a stable process receives at least $f_i + 1$ occurrences of this suspicion and may adopt it. This ensures the satisfaction of the *strong Byzantine completeness* property of the $\Diamond\mathcal{S}^M(Byz, \mathcal{A})$ FD.

Mistake Generation Let p_i be a process that has been suspected of not sending a message m. If eventually a stable process p_j receives m properly signed from p_i, p_j will declare a *mistake* on the suspicion and will spread m to the remaining nodes, so that they can do the same. In a network with Byzantine coverage, there will be at least one path formed only by stable processes between p_j and every stable process. Then, every other stable process will receive m and will be able to remove the related suspicion. This behavior allows a Byzantine process to provoke a suspicion and revoke it continuously, masking part of the omission failures and degrading the failure detector performance. Nevertheless, it is not possible to distinguish that situation from the slowness of a process or an instability on the channel.

Security Failure Detection In order to enable the *commision* (or *security*) failure detection, a message format must be established. Every message must also include a certificate that enables other processes to verify its coherence with algorithm $\mathcal{A}$. If a stable process detects the non validity of a received message, either for not obeying to the format or for a non valid justification, it will permanently suspect the sender and will forward the message to the remaining processes, so that the suspicion is propagated.

Notice that it is also necessary to detect *mutant messages*. This anomaly happens when a process sends two or more different versions of the same message. In their protocol, Kihlstrom *et al.* [KMMS03] deals with this problem by requiring stable processes to forward every received message. Moreover, processes should maintain a history of messages received by every process. Their model suppose, though, a point-to-point communication. In our model, processes communicate only through local broadcast under fair-lossy channels. Based on the recent advances and model propositions to implement reliable wireless channels [Koo04, PP05, BV05, BV07, BV10], we can assume that a message broadcast will be received with equal content by every stable process, so that it is impossible to send mutant messages.

4.2 Algorithm Description

Algorithms A1 and A2 implement a $\Diamond\mathcal{S}^M(Byz, \mathcal{A})$ FD. Every process p_i executes three parallel tasks, described below. The variables, primitives and procedures are described afterwards.

T1. Generating new SUSPICION messages (lines 5-14, A1). When algorithm $\mathcal{A}$ requires the processes to exchange a message m (line 6), every process p_i waits until the reception of m from at least α_i neighbors, whose identifiers are stored in the set rec_from_i (lines 7-8). For the remaining processes known by p_i, it adds an internal omission failure suspicion (lines 9-11). Then every message has its format and certificates verified through *ValidateReceived()* (lines 12-14, 30-41, A1). Incorrect messages lead to security failure suspicions (lines 34-35, A1) and update the detector output; correct messages generate mistakes on possible omission failure suspicions (lines 37-38, A1).

T2. Receiving SUSPICION messages and $\mathcal{A}$ messages from slow processes (lines 16-18, A1). When a message m is received from a remote process p_j, its format and certificates are verified through *ValidateReceived()*. There are

Algorithm 1 Byzantine Failure Detector (A1)

1: **init:**
2: $output_i \leftarrow known_i \leftarrow \emptyset$; $extern\_susp_i \leftarrow [][]$
3: $intern\_susp_i \leftarrow mistake_i \leftarrow []$; $byzantine_i \leftarrow \emptyset$
4:
5: **Task** T1: */* generating new suspicions */*
6: **when** p_i requires a message m **do**
7: **wait until** receive m properly signed for the first time from at least α_i distinct processes
8: $rec\_from_i \leftarrow \{p_j \mid p_i$ received a message from p_j at line 7$\}$
9: **for all** $p_j \in (known_i \setminus rec\_from_i)$ **do**
10: AddInternalSusp(p_j, m)
11: **end for**
12: **for all** m_j received at line 7 **from** p_j **do**
13: ValidateReceived(p_j, m_j)
14: **end for**
15:
16: **Task** T2: */* receiving the internal state of another process or messages from slow process */*
17: **upon receipt of** m properly signed **from** p_j **do**
18: ValidateReceived(p_j, m)
19:
20: **Task** T3: */* broadcasting suspicion state */*
21: **loop**
22: broadcast $\langle$SUSPICION, $byzantine_i$, $mistake_i$, $intern\_susp_i$, $extern\_susp_i\rangle$
23: **end loop**
24:
25: */* AUXILIARY PROCEDURES */*
26: **procedure** AddInternalSusp(q, m):
27: $intern\_susp_i[q] \leftarrow intern\_susp_i[q] \cup \{m.\text{id}\}$
28: $output_i \leftarrow output_i \cup \{q\}$
29:
30: **procedure** ValidateReceived(q, m):
31: **if** m was sent directly by q **then**
32: $known_i \leftarrow known_i \cup \{q\}$
33: **end if**
34: **if** m is not properly formed **or** m is not properly justified **then**
35: AddByzantine(q, m)
36: **else**
37: **if** $m.\text{id} \in intern\_susp_i[q]$ **or** m was forwarded **then**
38: AddMistake(q, m)
39: **end if**
40: UpdateSuspicions(q, m)
41: **end if**

two possibilities: (1) m is a message required by $\mathcal{A}$ that is received lately by p_i, probably after a suspicion over p_j has been generated in task T1. In this case, m is treated similarly to task T1 (lines 30-38, A1). (2) m is a SUSPICION message. In this case, the internal state of p_i is going to be updated (line 40, A1) as follows:

Updating internal state (lines 13–35, A2). Upon the receipt of a SUSPICION message from a neighbor q (line 19, A2), a process p_i updates its internal state with new information. Internal and external suspicions from q are added to the external suspicion set of p_i (lines 15-26, A2), possibly generating new internal suspicions (lines 37-41, A2). Note that a security failure suspicion will be raised on q if the SUSPICION message m is malformed or unjustified. Mistake information and security failure proofs are treated similarly to messages received directly from the sender through *ValidateReceived()*.

T3. Broadcasting suspicions and mistakes (lines 20-23, A1). This task periodically sends SUSPICION messages to p_i's neighbors carrying out its view on internal and external suspicions, mistakes and security failure proofs. The neighbors of p_i will receive that message in task T2.

VARIABLES:
• $output_i$: stores the failure detector output, i.e., the set of processes identities that p_i suspects of having failed;
• $known_i$: stores the set of processes that have communicated with p_i, i.e., its neighborhood. It is updated at the reception of SUSPICION messages or messages required by $\mathcal{A}$;
• $extern\_susp_i$: matrix that stores external suspicions (generated by other processes). The matrix is indexed by a process identifier q and a message identifier idm. Every entry stores the set of processes from which p_i has received suspicions about q and message(idm);
• $intern\_susp_i$: array of internal suspicions. An internal suspicion is generated by not receiving a message required by $\mathcal{A}$ or by the presence of at least $f_i + 1$ external suspicions on a pair process-message;
• $mistake_i$: array that stores, for every applicable process p_j, the set of mistakes related to p_j. A mistake is stored as a message required by $\mathcal{A}$ about which a suspicion has been raised;
• $byzantine_i$: set of tuples in the form ⟨process, message⟩ that prove Byzantine behavior on the related process. The notation $\langle p, - \rangle$ means "any tuple related to process p";
• rec_from_i: set of processes from which p_i received the message required by $\mathcal{A}$.

PRIMITIVES:
• m.id – returns the identifier of message m;
• message(idm) - returns the message bound to identifier idm;
• broadcast m – broadcasts a message m to the neighbors of p_i;
• keys(v) – returns the index set of a dynamic array v;
• ids(s) – returns the set of identifiers bound to the messages at set s.

AUXILIARY PROCEDURES:
• *AddInternalSusp*(q, m) (lines 26-28, A1): adds an internal suspicion on process q and message m;

Algorithm 2 Byzantine Failure Detector (A2)

```
1: procedure AddByzantine(q, m):
2: output_i ← output_i ∪ {q};
3: byzantine_i ← byzantine_i ∪ {⟨q, m⟩}
4:
5: procedure AddMistake(q, m):
6: mistake_i[q] ← mistake_i[q] ∪ {m}
7: extern_susp_i[q][m.id] ← ∅
8: intern_susp_i[q] ← intern_susp_i[q] \ {m.id}
9: if intern_susp_i[q] = ∅ and ∄⟨q, −⟩ ∈ byzantine_i then
10:    output_i ← output_i \ {q}
11: end if
12:
13: procedure UpdateSuspicions(q, m):
14: if m = ⟨SUSPICION, byzantine_q, mistake_q, intern_susp_q, extern_susp_q⟩ then
15:    for all p_x ∈ keys(extern_susp_q) do
16:       for all idm_x ∈ keys(extern_susp_q[p_x]) properly signed | idm_x ∉
          ids(mistake_i[p_x]) do
17:          for all p_y ∈ extern_susp_q[p_x][idm_x] do
18:             AddExternalSusp(p_x, idm_x, p_y)
19:          end for
20:       end for
21:    end for
22:    for all p_x ∈ keys(intern_susp_q) do
23:       for all idm_x ∈ intern_susp_q[p_x] properly signed | idm_x ∉ ids(mistake_i[p_x])
          do
24:          AddExternalSusp(p_x, idm_x, q)
25:       end for
26:    end for
27:    for all p_x ∈ keys(mistake_q) do
28:       for all m_x ∈ mistake_q[p_x] properly signed do
29:          ValidateReceived(p_x, m_x)
30:       end for
31:    end for
32:    for all ⟨p_x, m_x⟩ ∈ byzantine_q | m_x is properly signed do
33:       ValidateReceived(p_x, m_x)
34:    end for
35: end if
36:
37: procedure AddExternalSusp(q, idm, p_s):
38: extern_susp_i[q][idm] ← extern_susp_i[q][idm] ∪ {p_s}
39: if |extern_susp_i[q][idm]| ≥ f_i + 1 then
40:    AddInternalSusp(q, message(idm))
41: end if
```

• *ValidateReceived*(q, m) (lines 30-41, A1): verifies if the message m received from q is valid (well-formed and justified), removing any suspicions related to the pair (q, m) in the affirmative case, otherwise generating a security failure suspicion. Also, updates the set of nodes known by p_i ($known_i$) and forwards the messages to *UpdateSuspicions*();
• *UpdateSuspicions*(q, m) (lines 13–35, A2): if m is of the type SUSPICION, updates the internal state of p_i with the information in m.
• *AddByzantine*(q, m) (lines 1-3, A2): adds q permanently to the list of Byzantine processes (and, consequently, to the FD output), along with the message m as a proof of the Byzantine failure;
• *AddMistake*(q, m) (lines 5-11, A2): adds a mistake on a previous suspicion about process q and message m, removing any corresponding internal or external suspicions. If q has no other suspicions and has not presented Byzantine behavior, removes q from the failure detector output;
• *AddExternalSusp*(q, idm, p_s) (lines 37-41, A2): adds an external suspicion from p_s about process q and message identified by idm. Also, if there are at least $f_i + 1$ external suspicions about q and message(idm), generates a corresponding internal suspicion, if not already present.

5 Correctness Proof

To implement a failure detector of class $\Diamond\mathcal{S}^M(Byz, \mathcal{A})$, the algorithm in Section 4 should satisfy the *Byzantine strong completeness* and the *eventual weak accuracy* properties. In the following, a sketch of the proofs of the algorithm is given.

5.1 Byzantine Strong Completeness

Lemma 1 *If a process p_i never send message m, then a stable known process will never execute* AddMistake(p_i, m).

Proof 1 *Assume, by contradiction, that some stable known process p_j executes AddMistake(p_i, m). Notice that AddMistake() is only invoked into the procedure ValidateReceived() (line 13, A2). The procedure ValidateReceived() is for its turn invoked in 3 cases: (1) on the reception of messages required for $\mathcal{A}$ on task T1 (line 13, A1) and task T2 (line 18, A1); (2) on the reception of* SUSPICION *messages on task T2 (line 18, A1); (3) on the update of the internal state with information from the neighbors (execution of UpdateSuspicions(), lines 29 and 33, A2). In all cases, the authentication of message m is properly verified (lines 7 and 17, Alg 1; lines 28 and 32, A2). From this fact and since channels are reliable, a faulty process p_f cannot send m in the place of p_i. The occurrence of case (2), specifically, could not lead to a call to AddMistake(), since there is no suspicion related to messages* SUSPICION. *Moreover, correct* SUSPICION *messages are not forwarded.*

Finally, we conclude that in all cases there is a contradiction, since for m to be received, p_i should had sent it at some point in time. Thus, the lemma follows.

Lemma 2 *Let p_i be an omission faulty process. Then, eventually, every stable known process $p_j \in \Pi$ will permanently include p_i in its $output_j$ set.*

Proof 2 *Let m be the first message required by $\mathcal{A}$ and not sent by p_i. Let t be the moment in which $\mathcal{A}$ requires m from p_i. Let u be the first moment at which $|N_i^u| \geq 2f_i + 1$. This predicate holds due to the Byzantine coverage Assumption 1(2). It is true that $t \geq u$, because, before u, p_i was not in the run. Two cases are possible.*

CASE 1*: $p_j \in N_i^t$. If this happens then p_j has received a message of type* SUSPICION *from p_i before time t. Thus, $p_i \in known_j$, according to the execution of lines 17-18, 31-33, A1. Whenever the execution of $\mathcal{A}$ requires m, p_j will wait until the reception of m from α_j distinct processes (lines 6-7, A1). This predicate will be satisfied at some point in time, since at most f_j process are faulty and $|N_j| > 2f_j$ (Assumption 1(2)). Since p_i did not send m, p_i will not be included in rec_from_j (line 8, A1). According to the execution of lines 9-11 and 27-28, A1, $m.id$ will be included in $intern\_susp_j[p_i]$ and p_i will be included in $output_j$. Since p_i is faulty, it will never send m afterwards. Thus, from Lemma 1 and lines 8-11, A2, $m.id$ will never be removed from $intern\_susp_j[p_i]$ and from p_i de $output_j$.*

CASE 2*: If $p_j \notin N_i^t$. Since the network has Byzantine coverage (Assumption 1(1)), then there is at least a path P, between p_j and each stable known process $p_k \in N_i^t$ composed only by stable known processes. If there is more than one path, than take the one with minimum distance. Let us prove, by induction on the length of P, that, eventually, p_k is added to $extern\_susp_j[p_i][m.id]$.*
(1) If $|P| = 1$, then p_j is a neighbor of p_k. In this case, at some point in time, p_k send a message SUSPICION *s (line 22, A1) with the certified information that $m.id \in intern\_susp_k[p_i]$; since channels are reliable, at some point, p_j receives s (line 17, A1). Since p_k is stable known, s is duly certified, formed and justified; from Lemma 1, $m \notin mistake_j[p_i]$; Thus, from lines 18, 40 A1 and lines 22-26, A2, p_k is added to $extern\_susp_j[p_i][m.id]$ in line 38, A2 and the affirmation holds.*
(2) If $|P| > 1$, we can assume by induction that the affirmation is true for the path $P - p_j$ between p_k and a stable known process p_l, such that p_l and p_j are neighbors. For induction hypothesis, eventually, p_k is added to $extern\_susp_l[p_i][m.id]$ and, afterwards, p_l sends a message SUSPICION *s (line 22, A1) with this information certified by p_k. Since channels are reliable, eventually p_j receives s (in line 17, A1). Since p_l is stable known, s is duly certified, formed and justified; from Lemma 1, $m \notin mistake_j[p_i]$; thus, from lines 18, 40, A1 and lines 15-21, A2, p_k is added to $extern\_susp_j[p_i][m.id]$ in line 38, A2 and the affirmation holds.*

From the above conditions, from $|N_i^t| \geq 2f_i + 1$ and knowing that there is at most f_i faulty processes, it follows that, at some point in time, p_j executes line 40, A2 and, from lines 27-28, A,1 it adds $m.id$ to $intern\_susp_j[p_i]$ and p_i to $output_j$. Again, from Lemma 1, it follows that p_i will never be removed from $output_j$.

Lemma 3 *Let p_i be a commission faulty process. Then, eventually, every stable known process $p_j \in \Pi$ will permanently include p_i in its $output_j$ set.*

Proof 3 *A commission faulty (or security faulty) is produced when p_i sends a message m not in accordance with $\mathcal{A}$. In this case, m is not well formed or not justified. Notice that, due to the adoption of a broadcast communication pattern, mutant messages are not possible. Moreover, m is a certified message; otherwise, an undiagnosable faulty had been produced.*

Since the network has Byzantine coverage (Assumption 1(1)), then there is a path P between p_i and each stable known process p_j composed only by stable known processes, except for p_i. If there is more than one path, than take the one with minimum distance. Let us prove, by induction on the length of P, that, eventually, p_j adds $\langle p_i, m\rangle$ to $byzantine_j$ and p_i to $output_j$.
(1) If $|P| = 1$, then $p_j \in range_i$ and, since channels are reliable and m is certified, p_j receives m at some moment in lines 7 or 17, A1. In both cases, the procedure ValidateReceived() (lines 13 and 18, A1) is invoked. This procedure will attest the non-validity of m at line 34, A1. For its turn, the procedure AddByzantine() (lines 1-3, A2) adds p_i to $output_j$ and $\langle p_i, m\rangle$ to $byzantine_j$, and the affirmation holds.
(2) If $|P| > 1$, we can assume by induction that the affirmation is true for the path $P - p_j$ between p_i and a stable known process p_k, such that p_k and p_j are neighbors. In this case, $\langle p_i, m\rangle$ is in $byzantine_k$ and, at some point in time, p_k send a message SUSPICION *s with this information (line 22, A1); since channels are reliable, at some moment, p_j receives s at line 17, A1. Since p_l is stable known, s is duly certified, formed and justified; thus, as m is certified from lines 18, 40 A1 and lines 32-34, A2, p_j invokes the procedure ValidateReceived() and attest the non-validity of m; thus, p_i is added to $output_j$ and $\langle p_i, m\rangle$ to $byzantine_j$ and the affirmation holds.*

From the above conditions and since p_j only removes p_i from $output_j$ if there is no pair $\langle p_i, -\rangle$ in $byzantine_j$ (lines 9-11, A2), p_i is definitely added to $output_j$ and the lemma follows.

5.2 Eventual Weak Accuracy

Lemma 4 *If p_i and p_j are stable known processes, then, during the run, p_j never invokes the procedure* AddByzantine$(p_i, -)$.

Proof 4 *Notice that the only invocation of AddByzantine() is in line 35, A1 into the procedure ValidateReceived(). From line 34, A1 knowing that p_j is stable known, this calling only occurs if p_i has sent a message which was not in good format or not justified; but this is impossible, since p_i is stable known. If a faulty process sends such a message in the place of p_i, then process p_j will discard it. This happens because channels are reliable and p_j validates the authentication of every message it receives (lines 7 and 17, A1 and lines 28 and 32, A2), and the lemma follows.*

Lemma 5 *Let p_i be a stable known process and m be a message required by $\mathcal{A}$. If every node in N_i receives m from p_i in line 7, A1, then no stable known process $p_j \in \Pi$ will invoke* AddInternalSusp (p_i, m).

Proof 5 *The procedure AddInternalSusp() is called in 2 situations:* (1)*: in the task T1, during the reception of messages from $\mathcal{A}$ (line 10, A1);* (2)*: in the procedure AddExternalSusp() (line 40, A2), when the process receives more than f external suspicions regarding p_i.*

CASE 1: *From the lemma hypothesis, nodes in N_i receive m from p_i and then add p_i to their rec_from set in line 8, A1. Thus, they do not invoke AddInternalSusp(p_i, m) in line 10, A1. A process p_j out of N_i cannot receive messages directly from p_i, thus, it will never add p_i to $known_j$ (lines 31-33,*

A1); thus, it will never invoke AddInternalSusp (p_i, m) *in line 10, A1. Both situations confirm Case 1.*

CASE 2: *Notice that AddExternalSusp() is only invoked in lines 18 and 22, A2. Since a stable known process only updates its extern_susp set on the execution of AddExternalSusp(), every external suspicion regarding a stable known process was firstly generated as an internal suspicion (see lines 39 and 40, A2). From the same argument of Case (1), a stable known process* p_j *never adds* $m.id$ *to* $intern\_susp_j[p_i]$ *on the execution of task T1. If a Byzantine process* p_k *adds* p_j *to* $extern\_susp_k[p_i][m.id]$*, then a stable known process will not adopt this suspicion since the authentication of the message is verified in line 16, A2. A faulty process* p_j *can otherwise add* $m.id$ *to* $intern\_susp_j[p_i]$ *and certify this information. Nonetheless, there are at most* f_i *faulty processes and the predicate in line 27, A2 is never satisfied. Thus, no stable known process will invoke AddInternalSusp* (p_i, m) *in line 40, A2. The lemma, thus follows.*

Lemma 6 *Let* p_i *be a stable known process. If there is a message* m *and a stable known process* p_j *such that* $m.\text{id} \in intern\_susp_j[p_i]$ *during the run, then, eventually, process* p_j *will invoke* AddMistake (p_i, m).

Proof 6 *Two cases are possible.*
CASE 1*: Process* $p_j \in range_i$*. Since* p_i *is stable known, eventually* p_j *receives* m *from* p_i *(duly certified, formed and justified) (line 17, A1). From the hypothesis of lemma,* $m.id \in intern\_susp_j[p_i]$*, thus, from lines 18, 34, 37 A1, since* p_i *is stable known,* p_j *will call AddMistake* (p_i, m) *in line 38, A1.*

CASE 2*: Process* $p_j \notin range_i$*. By a similar argument used in Lemma 5,* $p_i \notin known_j$*. Thus, other stable known process* $p_k \in range_i$ *raises the suspicion; that is, there is a* $p_k \in range_i$ *such that* $m.id \in intern\_susp_k[p_i]$ *Since the network has Byzantine coverage (Assumption 1(1)), then there is a path* P *between* p_k *and* p_j *composed only by stable known processes. If there is more than one path, than take the one with minimum distance. Let us prove, by induction on the length of* P*, that, eventually, each* p_l *in* P *invokes AddMistake* (p_i, m)*, and thus* $m \in mistake_l[p_i]$.
(1) If $|P| = 0$*,* P *has only* p_k *and for the same argument of Case (1), the affirmation holds.*
(2) If $|P| > 0$*, we can assume by induction that the affirmation is true for the path* $P - p_j$ *between* p_k *and* p_l*, and that at some moment,* $m \in mistake_l[p_i]$*. Afterwards,* p_l *broadcast a message* SUSPICION s *with* m *duly certified in* $mistake_l[p_i]$*. Since channels are reliable, at some point in the future,* p_j *receives* s *in line 17, A1. Since* p_l *is stable known,* s *is duly certified, formed and justified. Thus, for the execution of line 18, A1 and lines 14 and 27-31, A2,* p_j *calls ValidateReceived* (p_i, m)*. Since* p_i *is stable known,* m *is duly certified, formed and justified. Since* m *was forwarded by* p_l*,* p_j *calls AddMistake* (p_i, m) *in line 38, A1 and the affirmation holds. The lemma thus follows.*

Lemma 7 *Let* p_i *be a stable known process that satisfies* $Byz\mathcal{RP}$ (p_i)*. Eventually, every stable known process* $p_j \in \Pi$ *is such that* $p_i \notin output_j$.

Proof 7 *From Lemma 4, we can attest that* p_i *will never be added to* $output_j$ *in line 2, A2. From property* $Byz\mathcal{RP}$ (p_i)*, there exists a time* t *after which every message* m *required by* $\mathcal{A}$ *in* p_i *is received by the neighbors of* p_i *in line 7, A1. From Lemma 5, we can attest that* p_j *does not add* p_i *to* $output_j$ *in*

a call to AddInternalSusp (p_i, m). *For every message* m' *required by* $\mathcal{A}$ *before* t, *it is possible that* $m' \in intern\_susp_j[p_i]$. *But, from Lemma 6, at some point in the future,* p_j *calls AddMistake* (p_i, m'); *thus, for line 8, A2, eventually* $intern\_susp_j[p_i] = \emptyset$. *From Lemma 4, there is no pair* $\langle p_i, - \rangle$ *in* $byzantine_j$; *thus,* p_i *is removed from* $output_j$ *in line 10, A2 and the lemma holds.*

Theorem 1 *Algorithms 1 and 2 implement a Byzantine failure detector of class* $\Diamond\mathcal{S}^M(Byz, \mathcal{A})$.

Proof 8 *The theorem follows from Lemma 2, 3 and 7 and from the specification of class* $\Diamond\mathcal{S}^M(Byz, \mathcal{A})$.

6 Conclusion

This paper presented a Byzantine failure detector of class $\Diamond\mathcal{S}^M(Byz, \mathcal{A})$ with two innovative features that favor the scalability and adaptability: (i) it is suitable for dynamic distributed systems in which the membership is unknown and (ii) it does not rely on timers to detect omission failures. As a future work, we plan to (i) extend the protocol to tolerate node mobility, (ii) implement the protocol for performance evaluation, and (iii) prove (or find a counterexample for) the impossibility of detecting Byzantine failures in a time-free manner with a non symmetrical $(1 \rightarrow n)$ communication.

References

[Agu04] Marcos K. Aguilera. A Pleasant Stroll through the Land of Infinitely Many Creatures. *ACM SIGACT News*, 35(2):36–59, June 2004.

[AHNRR02] Baruch Awerbuch, David Holmer, Cristina Nita-Rotaru, and Herbert Rubens. An on-demand secure routing protocol resilient to byzantine failures. In *Proc. of the 1st ACM Workshop on Wireless Security*, pages 21–30, New York, NY, USA, 2002. ACM.

[BHRT03] R. Baldoni, J. Hélary, M. Raynal, and L. Tanguy. Consensus in Byzantine asynchronous systems. *J. Discrete Algorithms 1(2): 185-210 (2003)*, 1 2003.

[BV05] V. Bhandari and N. H. Vaidya. On reliable broadcast in a radio network. In *Proc. of the 24th annual ACM symposium on Principles of distributed computing*, pages 138–147. ACM, 2005.

[BV07] Vartika Bhandari and Nitin H. Vaidya. Reliable local broadcast in a wireless network prone to byzantine failures. In *DIALM-POMC*, 2007.

[BV10] Vartika Bhandari and Nitin H. Vaidya. Reliable broadcast in radio networks with locally bounded failures. *IEEE Transactions on Parallel and Distributed Systems*, 21:801–811, 2010.

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[Dou02] John R. Douceur. The sybil attack. In *Revised Papers from the First Int. Workshop on Peer-to-Peer Systems*, pages 251–260, London, UK, 2002. Springer-Verlag.

[DS98] Assia Doudou and André Schiper. Muteness detectors for consensus with byzantine processes. In *PODC '98: Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*, page 315, New York, NY, USA, 1998. ACM.

[HK09] Andreas Haeberlen and Petr Kuznetsov. The Fault Detection Problem. In *Proceedings of the 13th International Conference on Principles of Distributed Systems (OPODIS'09)*, December 2009.

[HKD07] Andreas Haeberlen, Petr Kuznetsov, and Peter Druschel. PeerReview: Practical accountability for distributed systems. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)*, Oct 2007.

[JAF06] Ernesto Jiménez, Sergio Arévalo, and Antonio Fernández. Implementing unreliable failure detectors with unknown membership. *Inf. Process. Lett.*, 100(2):60–63, 2006.

[KMMS03] Kim Potter Kihlstrom, Louise E. Moser, and P. M. Melliar-Smith. Byzantine Fault Detectors for Solving Consensus. *The Computer Journal*, 46(1):16–35, January 2003.

[Koo04] Chiu-Yuen Koo. Broadcast in radio networks tolerating byzantine adversarial behavior. In *PODC '04: Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 275–282, New York, NY, USA, 2004. ACM.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[MMR03] Achour Mostefaoui, Eric Mourgaya, and Michel Raynal. Asynchronous Implementation of Failure Detectors. In *Proc. of the 2003 Int. Conf. on Dependable Systems and Networks*, page 351, Los Alamitos, CA, USA, 2003. IEEE Computer Society.

[MR97] D. Malkhi and M. Reiter. Unreliable intrusion detection in distributed computations. In *Proc. 10th Computer Security Foundations Workshop*, pages 116–124, Rockport, MA, 1997. IEEE Computer Society Press, Los Alamitos, CA.

[MRT+05] Anchour Mostefaoui, Michel Raynal, Corentin Travers, Stacy Patterson, Divyakant Agrawal, and Amr El Abbadi. From Static Distributed Systems to Dynamic Systems. In *Proc. of the 24th IEEE Symp. on Reliable Distributed Systems*, pages 109–118, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[PP05] Andrzej Pelc and David Peleg. Broadcasting with locally bounded byzantine faults. *Inf. Process. Lett.*, 93(3):109–115, 2005.

[SAB+08] Pierre Sens, Luciana Arantes, Mathieu Bouillaguet, Véronique Simon, and Fabíola Greve. An unreliable failure detector for unknown and mobile networks. In *Proc. of the 12th Int. Conf. on Principles of Distributed Systems*, pages 555–559, Berlin, Heidelberg, 2008. Springer-Verlag.

[Sch96] Bruce Schneier. *Applied Cryptography (2nd ed.)*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

Centre de recherche INRIA Paris – Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399