Byzantine Failure Detection for Dynamic Distributed Systems

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

INRIA

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 — 20 pages

Abstract: Byzantine failure detectors provide an elegant abstraction for solving security problems. However, as far as we know, there is no general solution for this problem in a dynamic distributed system of unknown networks. This paper presents thus a first Byzantine failure detector for this context. The protocol has the interesting feature to be asynchronous, that is, the failure detection process does not rely on timers to make suspicions. This characteristic favors its scalability and adaptability and leads to an intriguing conjecture about the pattern of the overlying algorithm that uses the failure detector as a building block: it should be symmetrical.

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

* 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 défaillance Byzantins offrent une abstraction pour résoudre des problèmes de sécurité. 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. Elle induit aussi une curieuse conjecture sur le mode de communication de lŠalgorithme qui utilise le détecteur de défaillance comme brique de base : l'algorithme doit être symétrique.

Mots-clés : détecteurs de fautes, fautes Byzantines, systèmes répartis dynamiques, systèmes auto-organisant, détecteur de fautes asynchrones

1 Introduction

Modern distributed systems, structured over Mobile Ad-Hoc Networks (MANETs), sensor and P2P networks are inherently dynamic [1]. They are composed by a dynamic population of nodes, which randomly join and leave the network, at any moment of the execution. Usually, these systems present the following restrictions: (i) the message transmission latency is unpredictable; (ii) the network is not fully connected; (iii) there is an absence of centralized entities; (iv) it is not possible to provide nodes with a global view of the network topology, so that each node has a partial knowledge of the system composition; and (v) the nodes may change their location on the network. 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.

Security is a major problem on dynamic distributed systems. The dynamic population of nodes and the use a wireless network or the Internet as a communication media favors the action of malicious agents on the system. The Byzantine failure model [2] deals with 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. For example, a Byzantine process could try to assume the identity of another process, send incorrect values, duplicate messages or just do not send messages required by the protocol under execution. Thus, Byzantine fault tolerance plays an important role on the development of dependable dynamic distributed systems.

The unreliable failure detector abstraction [3] (or FD) provides a modular approach to deal with failures on asynchronous systems. It exempts the overlying protocol to deal with the failure treatment and synchrony requirements, so that it can just take care about its inherent task. For example, the consensus problem [4] summarizes many distributed agreement problems but cannot be solved in a system without synchrony requirements, even if one single process crashes [5]. If the system is augmented, though, with a $\Diamond\mathcal{S}$ failure detector [3], the consensus with crash failures can be solved without managing directly the synchrony requirements.

Most FD implementations suppose a static network with a previously known composition of nodes [6]. Some recent work has been proposed for the implementation of FD on mobile and self-organizing networks [7, 8], the former supposing a dynamic population, and the latter managing mobility. All of those protocols, though, rely on timers to make suspicions, i.e., a node monitors the others through the exchange of heartbeat messages. This strategy is not very adequate for a dynamic system due to its unpredictable message transmission latencies and frequent changes in topology.

Asynchronous FD do not rely on timers to make suspicions. This concept has been suggested by Mostefaoui *et al.* but for the context of static networks [9]. In [10], Sens *et al.* extended that purpose for systems with dynamic populations and node mobility. All of those works, though, deal only with crash failures, in this case, processes behave in a benign manner until they fail and finish their execution. The almost totality of Byzantine FD suppose a static network of known participants. Some few exceptions are [11, 12], but they solve only a subset of the properties of the Byzantine failure detection problem. In a recent work [13], Haeberlen *et al.* present the system PeerReview and pro-

pose 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 some dynamic systems but it does not consider the case of unknown networks.

This paper studies the problem of Byzantine failure detection and has the following original features: (i) it is suitable for a dynamic distributed system with unknown participants; (ii) it is asynchronous: it does not rely on timers to detect progress failures. (iii) it raises a conjecture regarding the communication pattern required for asynchronous failure detection on a Byzantine environment. The FD protocol is based on the approach of Kihlstrom *et al.* [14] for the Byzantine failure detection and the approach of Sens *et al.* [10] for the asynchronous detection. Commission (or security) failures are detected through the use of a standard message format, including signatures and certificates. Omission failure detection is based on the message exchange pattern of the overlying algorithm, because Byzantine failures are defined as deviation from the behavior specified by the algorithm [15]. The asynchronous detection is possible due to the message exchange pattern, to the network topology assumptions and to certain behavioral properties followed by the nodes on the system. The protocol does not manage, though, node mobility. The asynchronous detection choice has put into evidence two important results. First, we conjecture to be impossible to design asynchronous Byzantine fault-tolerant distributed algorithms without a distributed message exchange pattern between processes. Second, we infer that the use of the local broadcast primitive as the communication standard, typical from wireless channels, simplifies the security failure management, so that some classes of failures (e.g., *mutant* messages) cannot occur.

The rest of the paper is structured as follows: Section 2 provides the system model. Section 3 provides a characterization of Byzantine failures. Sections 4-6 present the FD protocol, some behavioral properties the system must obey so that the protocol works correctly and its implementation. The correctness proofs may be found at Section 7. Finally, Section 8 concludes the paper and provides future work.

2 System Model

The distributed system is composed by a set $\Pi = \{p_1, p_2, \ldots, p_n\}$ of $n > 4$ processes. Nodes may join or leave the network randomly, at any moment of the execution. No restrictions are made about processor speeds, relative clock drifts or message transmission delays; that is, the system is *asynchronous*. Processes are subject to Byzantine failures; thus, nodes may behave in an arbitrary, even malicious, manner; yet the system is equipped with a message authentication mechanism. The expected maximum number of processes that may fail is denoted by f. Processes have no knowledge about Π or n. In this sense, the network is unknown. The processes communicate by local broadcast through a wireless communication network and they can make use of the broadcast facility of this communication medium to know one another. Thus, we consider that a process knows a subset of Π, composed with nodes with whom it previously communicated.

In order to simplify definitions and proofs, we consider the existence of a global time t, nonetheless t is unknown to the processes. Every process is

uniquely identified, and a faulty process cannot obtain more than one identifier. Thus, it is impossible to launch a *sybil attack* [16].

Channels are reliable, so that a message sent is eventually received by every correct process in the neighborhood of its sender. Moreover, channels do not duplicate, modify or create messages and every correct process signs its messages in an unforgeable manner. Implementations of reliable broadcast communication in wireless networks may be found at [17, 18].

The system is represented by an undirected dynamic graph $G = (V, E)$, where $V = \Pi$ and $\{p_i, p_j\} \in E$ if and only if p_i and p_j are in the transmission range of each other at time t (p_i and p_j are called *neighbors*).

Definition 1 (Range) *The* $range_i^t$ *of a node* p_i *is defined by the set* $range_i^t := \{p_j \in \Pi : (p_i, p_j) \in E \text{ at time } t\}$. *The* $range_i^t$ *corresponds to the neighborhood of* p_i *in* G *at* t, $|range_i^t|$ *is the degree of* p_i *in* G *and* $p_i \in range_j^t \Leftrightarrow p_j \in range_i^t$. *That is, the communication between the nodes is* symmetrical.

Definition 2 (Range density (d)) *The density* d *of* $G = (V, E)$ *at time* t *is the size of the smallest range in the network, that is:* $d := min\ \{|range_i^t| : i \in \{1, 2, ..., n\}\}$ *at time* t. d *is therefore the minimum degree in the graph* G *at time* t.

Definition 3 (Network with Byzantine f-coverage) *A communication network represented by a graph* $G(V, E)$ *has Byzantine* f*-coverage if and only if* G *is* $(f + 1)$*-connected and* $d \geq 2f + 1$.

A k-connected graph has k vertex-disjoint paths between every two vertices, what leads to the following observation:

Observation 1 *In a network with Byzantine* f*-coverage, despite the presence of* $f < n$ *faulty processes, there is at least one path formed exclusively by correct nodes between every two correct nodes.*

3 Byzantine Failures

Byzantine failures are identified by the meeting of two requirements: (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 or protocol. The first requirement may be addressed by two distinct techniques: *information redundancy* or *unforgeable digital signatures* [2]. The second requirement can be met by adding *certificates* to the messages, so that its content may be validated [14].

Figure 1 shows the Byzantine failure categorization by Kihlstrom *et al.* [14]. Two superclasses are distinguished: *detectable*, when the external behavior of a process provides evidence of the failure and *non-detectable*, otherwise. Non-detectable failures are grouped in *unobservable*, when other processes cannot perceive the occurrence of a failure (e.g., when a faulty process informs a parameter different from the supplied by the user) and *undiagnosable*, when it is not possible to identify the perpetrator of failure (e.g., the processes receive an unsigned message).

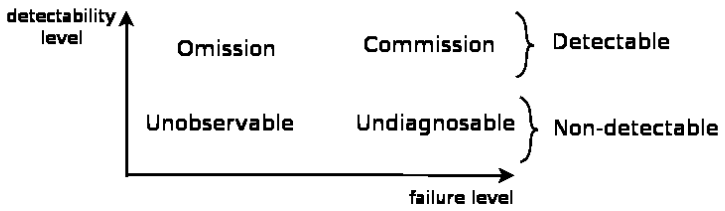

Figure 1: Byzantine failure categorization [14]

Detectable failures are classified in *progress* (or *omission*) failures and *security* (or *commission*) failures. Progress failures hampers the termination of the computation, since a faulty process does not send the messages required by its specification or sends it only to part of the system. Security 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 (a faulty process could, thus, send a message with different data to different processes); (ii) the messages sent must conform the algorithm under execution.

Failure Detector Properties. Kihlstrom *et al.* [14] define Byzantine failure detector classes which differ from those described by Chandra and Toueg [3], 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 $\Diamond\mathcal{S}(Byz, \mathcal{A})$, the focus of this work, is defined by the following properties:
(i) *Strong Byzantine completeness* (for algorithm $\mathcal{A}$): eventually, every correct process suspects permanently every process that has detectably deviated from $\mathcal{A}$;
(ii) *Eventual weak accuracy*: eventually, one correct process is never suspected by any correct process.

4 Basic Principles of an Asynchronous Byzantine Failure Detector

In this section the fundamental principles of our asynchronous Byzantine failure detector is presented.

4.1 Message Exchange Pattern

Most of the protocols for crash failure detection are based on the exchange of heartbeat messages. Nevertheless, in a Byzantine environment, due to the occurrence of malicious processes, such a mechanism is no longer enough. A faulty process may correctly answer the failure detector messages, yet without guaranteeing progress and safety to the algorithm under execution. Therefore, the failure detection must be based on the pattern of the messages sent during the execution of algorithm $\mathcal{A}$ that uses the failure detector. Thus, similarly to Kihlstrom's strategy [14], suspicions are raised in function of the messages required by $\mathcal{A}$. Nonetheless, differently from it, such a suspicions are identified

asynchronously, without the use of timers to detect omission failures. Moreover, the failure detection follows a local message exchange pattern, i.e., between the nodes in the neighborhood.

To implement the asynchronous failure detection, a process will wait until the reception of messages required by $\mathcal{A}$ from at least α distinct senders and it will suspect an omission faulty from the remaining processes. The detection follows a local message exchange pattern, i.e., between the nodes in the neighborhood [10]. Thus, α corresponds to the minimum amount of correct nodes in the neighborhood of a node in the system; that is $\alpha \geq (d-f)$ or more precisely, for a process p_i, $\alpha \geq (|range_i^t| - f)$ at some point t in time. In practice, its actual value depends on the type of dynamic network considered (either Manet, sensor or P2P network) as well as the current topology of the network during execution. Notice though that a faulty process may send RESPONSE messages without respecting the progress requirements of algorithm $\mathcal{A}$. Thus, the QUERY-RESPONSE message pattern on which Sen's protocol is based is not adequate to a Byzantine environment.

Our protocol, thus, raises suspicions based on the exchange of messages required by algorithm $\mathcal{A}$. So, a process will wait until the reception of messages required by $\mathcal{A}$ from at least α distinct senders and it will suspect an omission faulty from the remaining processes. It is important to notice that, in order to enable suspicions, the communication pattern followed by algorithm $\mathcal{A}$ must be distributed. That is, at every step, all nodes must exchange messages, following a $n \rightarrow n$ pattern. So, the protocol followed by $\mathcal{A}$ should be symmetrical. Since the detector uses a local message exchange pattern, this symmetrical communication must occur at least between processes in the same range. Symmetrical consensus algorithms based on a $\Diamond\mathcal{S}$ failure detectors have been proposed [19].

Another important point to consider is that the detection follows an asynchronous pattern. As suspicions are based on the message exchange pattern of the algorithm, we conjecture to be impossible to detect omission failures if such a pattern is in 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 [3]. 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 in an asynchronous manner, if the message exchange pattern on the algorithm $\mathcal{A}$ is $1 \rightarrow n$; that is, if algorithm $\mathcal{A}$ requires a single process to send messages to the remaining (n).*

Corollary 1 *The asynchronous mode of Byzantine failure detection may only be adopted by symmetrical protocols, on which all nodes execute the same role.*

4.2 Suspicion and Mistake Generation

Suspicion Generation. Every suspicion on a process p_i is related to a message m required by $\mathcal{A}$. Thus, messages must have unique identifiers. Suspicions are propagated on the network and a correct process will adopt a suspicion not generated by itself if and only if it receives it properly signed from at least

$f + 1$ different senders. This requirement, of at least $f + 1$ messages, denies a malicious process to impose suspicions on correct processes. Figure 2 shows this mechanism for a portion of the network, supposing $f = 1$. The neighbors of a process $p1$, faulty by omission, detect that $p1$ is not sending the messages it should. In a network with Byzantine f-coverage, at least two $(f + 1)$ neighbors of $p1$, in this case $p2$ and $p3$, are correct and shall spread a suspicion of failure S to their respective neighbors. There is a path formed only by correct processes between any correct process (e.g., $p10$) and $p2$ ($p10p9 \cdots p5p2$) and $p3$ ($p10p3$) (see Observation 1), so that it receives at least two $(f+1)$ occurrences of S and may adopt the suspicion.

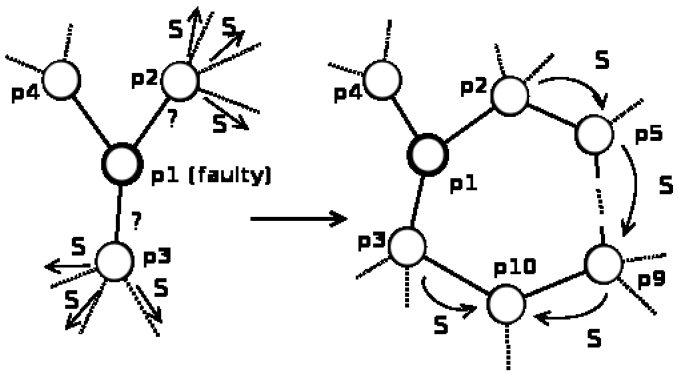

Figure 2: Suspicion generation on the proposed protocol ($f = 1$)

Mistake Generation. Let p_i be a process that has been suspected of not sending a message m. If eventually a correct process p_j receives m 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. At Figure 3, a slow process $p1$, about which a suspicion had been raised, sends the required message m (represented by the envelope) to a correct process $p2$. In a network with Byzantine f-coverage, there will be at least one path formed only by correct processes between $p2$ and every correct process. For example, for $p10$, we have the path $p2p5 \cdots p9p10$. Then $p10$ (or any other correct process) will receive m and will be able to remove the related suspicion.

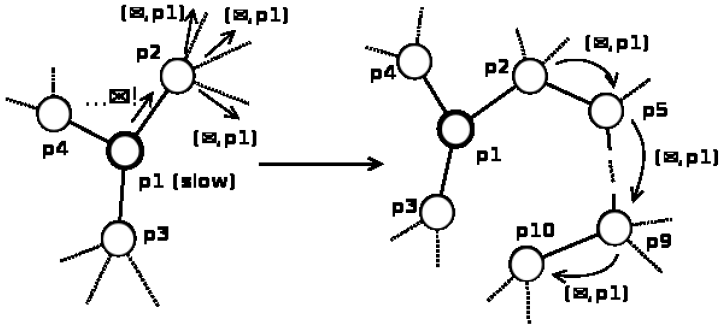

Figure 3: Mistake generation on the proposed protocol ($f = 1$)

This behavior allows a malicious 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 communication channel.

4.3 Security Failure Detection

In order to enable the 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 correct process detects the non validity of a received message, either for not obeying to the format or for incorrect justification, it will permanently suspect the sender and will forward the message to the remaining processes, so that the suspicion is propagated. Figure 4 represents that situation: a faulty process $p1$ commits a security failure by spreading a corrupted message m (represented by the asterisk). The failure is detected by the correct neighbor $p2$, who forwards the corrupted message to the remaining of the network. Since there is a path formed only by correct processes between $p2$ and every correct process (for $p10$, e.g., the path $p10p9 \cdots p5p2$), all of them will suspect permanently on $p1$.

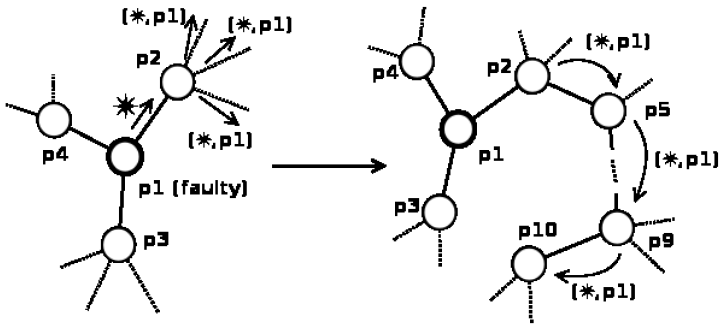

Figure 4: Security failure detection on the proposed protocol ($f = 1$)

In our protocol, suspicions, mistakes and security failure proofs are forwarded through a single message of the type SUSPICION, which does not need to be certified. Unsigned messages are discarded, as they configure an undiagnosable failure (see Section 3). Then, the detector does not commit errors in the security failure detection for algorithm $\mathcal{A}$.

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.* [14] deals with this problem by requiring correct processes to forward every received message. Moreover, processes should maintain a history of messages received by every proces Their model suppose, though, a point-to-point communication. In our model, processes communicate only through local broadcast in reliable channels. Thus, we can certainly suppose that a message broadcast will be received with equal content by every correct process, so that it is impossible to send mutant messages. This leads to the following conclusion.

Observation 2 *In a Byzantine failure environment, the communication by broadcast on reliable channels simplifies the security management, since the neighbors of a sender have a consistent view of the messages sent.*

5 Behavioral System Properties

In order to implement the Byzantine failure detector, in an asynchronous manner, some assumptions about the system behavior should be done.

Property 1 (Byzantine Membership Property ($Byz\mathcal{MP}$)) *Let t be some point in time and KB_i^t the set of nodes that received a* SUSPICION *message from p_i until t. A process p_i satisfies the Byzantine membership property $Byz\mathcal{MP}$ if:*

$$Byz\mathcal{MP}^t(p_i) := \exists t \geq 0 : |KB_i^t| \geq 2f + 1$$

This property ensures that a new process p_i will eventually be known by at least $2f + 1$ processes, from which at least $f + 1$ will be correct. Therefore, p_i has to communicate with at least $2f + 1$ nodes in its range, sending them a SUSPICION message by broadcast. Therefore, if p_i fails, eventually at least $f+1$ correct processes will suspect p_i and spread the suspicion to the remaining of the system, so that the Byzantine strong completeness property of $\Diamond\mathcal{S}(Byz, \mathcal{A})$ detector is satisfied. Note that such communication must be done before the next step of algorithm $\mathcal{A}$; before that, p_i must not be considered as participant of the computation. If a new process does not communicate with any other, it is impossible to satisfy weak completeness [20].

In order to satisfy eventual weak accuracy property of $\Diamond\mathcal{S}(Byz, \mathcal{A})$ class, there must exist a correct process p_i whose messages from some point on are always among the first α received by its neighbors, at every request of $\mathcal{A}$. Thus, eventually p_i will no longer be suspected by any correct process, and any previous suspicion will be revoked through mistake messages. Thus, the network must exhibit a correct process that satisfies the *Byzantine responsiveness* property, defined as:

Property 2 (Byzantine Responsiveness Property ($Byz\mathcal{RP}$)) *Let t and u be some points in time and $rec\_from_j^t$ the set of at least α processes from which p_j received the message required by $\mathcal{A}$ at its last step in execution until t. The $Byz\mathcal{RP}$ property of the correct process p_i is defined as:*

$$Byz\mathcal{RP}^t(p_i) := \exists u : \forall t > u, \forall p_j \in range_i^t, p_i \in rec\_from_j^t$$

6 Asynchronous Byzantine Failure Detector for Dynamic Systems

Algorithms 1 and 2 implement a $\Diamond\mathcal{S}(Byz, \mathcal{A})$ failure detector, as soon as the network satisfies *Byzantine f-coverage*, *Byzantine membership* and *Byzantine responsiveness* properties described previously, and the protocol executed by algorithm $\mathcal{A}$ is symmetrical. A sketch of the correctness proofs may be found at Section 7.

Every process executes three parallel tasks, described below. The variables, primitives and procedures used by each process p_i are described afterwards.

T1. Generating new suspicions (lines 1-1, Algorithm 1). When algorithm $\mathcal{A}$ requires the processes to exchange a message m (line 1), every process p_i waits until the reception of m from at least α neighbors, whose identifiers are stored in the set rec_from_i (lines 1-1). For the remaining processes known by p_i, it adds an internal omission failure suspicion (lines 1-1). Then every message has its format and certificates verified (lines 1-1, Algorithm 1, and 2-2, Algorithm 2). Incorrect messages lead to security failure suspicions (lines 2-2, Algorithm 2) and update the detector output; correct messages generate mistakes on possible omission failure suspicions (lines 2-2, Algorithm 2).

T2. Receiving SUSPICION messages and messages from slow processes (lines 1-1, Algorithm 1). Since the messages sent by a slow process may be received after suspicion generation on its neighbors, we need to identify such events separately, what is done by this task. The messages are treated similarly to task T1. The treatment of SUSPICION messages will be explained below.

T3. Broadcasting suspicions and mistakes (lines 1-1, Algorithm 1). This task is executed periodically to send to p_i's neighbors its view on (internal and external) suspicions, mistakes and security failure proofs. The neighbors of p_i will receive that message in task T2 and will treat it as follows:

Updating internal state (lines 15 and 18-40, Algorithm 2). Upon the receipt of a SUSPICION message from a neighbor q (line 19), 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 20-31), possibly generating new internal suspicions (lines 32-34, Algorithm 1). Note that a security failure suspicion will be raised on q (lines 9-10) 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 (lines 34 e 38). Some optimizations in this procedure had been removed from this version of the protocol, in order to simplify demonstrations. The interested reader may find them at [21].

VARIABLES:
• $output_i$: stores the failure detector output, i.e., the set of identifiers of processes 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 o 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 $f + 1$ 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

Algorithm 1 Asynchronous Byzantine Failure Detector for Dynamic Distributed Systems

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 distinct processes
8: $rec\_from_i \leftarrow \{p_j \mid p_i$ received a message from p_j at line 1$\}$
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 1 **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** AddExternalSusp(q, idm, p_s):
31: $extern\_susp_i[q][idm] \leftarrow extern\_susp_i[q][idm] \cup \{p_s\}$
32: **if** $|extern\_susp_i[q][idm]| \geq f + 1$ **then**
33: AddInternalSusp(q, message(idm))
34: **end if**
35:
36: **procedure** AddMistake(q, m):
37: $mistake_i[q] \leftarrow mistake_i[q] \cup \{m\}$
38: $extern\_susp_i[q][m.\text{id}] \leftarrow \emptyset$
39: $intern\_susp_i[q] \leftarrow intern\_susp_i[q] \setminus \{m.\text{id}\}$
40: **if** $intern\_susp_i[q] = \emptyset$ **and** $\nexists\langle q, -\rangle \in byzantine_i$ **then**
41: $output_i \leftarrow output_i \setminus \{q\}$
42: **end if**

Algorithm 2 Asynchronous Byzantine Failure Detector for Dynamic Distributed Systems (continuing)

```
1: procedure AddByzantine(q, m):
2: output_i ← output_i ∪ {q};
3: byzantine_i ← byzantine_i ∪ {⟨q, m⟩}
4:
5: procedure ValidateReceived(q, m):
6: if m was sent directly by q then
7:   known_i ← known_i ∪ {q}
8: end if
9: if m is not properly formed or m is not properly justified then
10:  AddByzantine(q, m)
11: else
12:  if m.id ∈ intern_susp_i[q] or m was forwarded then
13:    AddMistake(q, m)
14:  end if
15:  UpdateSuspicions(q, m)
16: end if
17:
18: procedure UpdateSuspicions(q, m):
19: if m = ⟨SUSPICION, byzantine_q, mistake_q, intern_susp_q, extern_susp_q⟩
    then
20:  for all p_x ∈ keys(extern_susp_q) do
21:    for all idm_x ∈ keys(extern_susp_q[p_x]) properly signed | idm_x ∉
       ids(mistake_i[p_x]) do
22:      for all p_y ∈ extern_susp_q[p_x][idm_x] do
23:        AddExternalSusp(p_x, idm_x, p_y)
24:      end for
25:    end for
26:  end for
27:  for all p_x ∈ keys(intern_susp_q) do
28:    for all idm_x ∈ intern_susp_q[p_x] properly signed | idm_x ∉
       ids(mistake_i[p_x]) do
29:      AddExternalSusp(p_x, idm_x, q)
30:    end for
31:  end for
32:  for all p_x ∈ keys(mistake_q) do
33:    for all m_x ∈ mistake_q[p_x] properly signed do
34:      ValidateReceived(p_x, m_x)
35:    end for
36:  end for
37:  for all ⟨p_x, m_x⟩ ∈ byzantine_q | m_x is properly signed do
38:    ValidateReceived(p_x, m_x)
39:  end for
40: end if
```

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;
• verifying if a message is well-formed, justified (certified content) and signed;
• request by $\mathcal{A}$ of a message;
• 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 1-1, Algorithm 1): adds an internal suspicion on process q and message m;
• AddExternalSusp(q, idm, p_s) (lines 1-1, Algorithm 1): adds an external suspicion from p_s about process q and message identified by idm. Also, if there are at least $f + 1$ external suspicions about q and message(idm), generates a corresponding internal suspicion, if not already present;
• AddMistake(q, m) (lines 1-1, Algorithm 1): 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;
• AddByzantine(q, m) (lines 2-2, Algorithm 2): 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;
• ValidateReceived(q, m) (Algorithm 2, lines 2-2): 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 the following procedure, in order to update the suspicion state;
• UpdateSuspicions(q, m) (Algorithm 2, lines 18-40): if m if of the type SUSPICION, updates the internal state of p_i with the information in m.

7 Correctness Proof

To implement a failure detector of class $\Diamond\mathcal{S}(Byz, \mathcal{A})$, the algorithm in Section 6 should satisfy the *Byzantine strong completeness* and the *eventual weak accuracy* properties. In the following, a sketch of the correctness proofs of the algorithm is given.

7.1 Byzantine Strong Completeness

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

Proof: Assume, by contradiction, that some correct process p_j executes AddMistake(p_i, m). Notice that AddMistake() is only invoked into the procedure ValidateReceived()

(line 13, Alg. 2). 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, Alg. 1) and task T2 (line 18, Alg. 1); (2) on the reception of SUSPICION messages on task T2 (line 18, Alg. 1); (3) on the update of the internal state with information from the neighbors (execution of UpdateSuspicions(), lines 34 e 38, Alg. 2). In all cases, the authentication of message m is properly verified (lines 7 and 17, Alg 1; lines 33 and 37, Alg. 2, in respective). 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, at some point in time, every correct process $p_j \in \Pi$ will permanently include p_i in its $output_j$ set.*

Proof: 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 $|KB_i^u| \geq 2f+1$, knowing that KB_i^u comes from property 1. In fact, instant u exists due to the satisfaction of $Byz\mathcal{MP}$. It is true that $t \geq u$, because, before u, p_i was not in the run (see Section 5). Two cases are possible.

CASE 1: $p_j \in KB_i^t$. 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, Alg.1 and 6-8, Alg. 2. Whenever the execution of $\mathcal{A}$ requires m, p_j will wait until the reception of m from α distinct processes (lines 6-7, Alg. 1). This predicate will be satisfied at some point in time, since at most f process are faulty and $|range_j| \geq d$. Since p_i did not send m, p_i will not be included in rec_from_j (line 8, Alg. 1). According to the execution of lines 9-11 and 27-28, Alg. 1, 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 39-42, Alg. 1, m.id will never be removed from $intern\_susp_j[p_i]$ and from p_i de $output_j$.

CASE 2: If $p_j \notin KB_i^t$. Since the network has Byzantine f-coverage, then there is at least a path P, between p_j and each correct process $p_k \in KB_i^t$ composed only by correct 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, at some point in time, p_k is added to $extern\_susp_j[p_i][m.\text{id}]$.
(1) If $|P| = 1$ (P has length 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, Alg. 1) with the certified information that $m.\text{id} \in intern\_susp_k[p_i]$; since channels are reliable, at some point, p_j receives s (in line 17, Alg. 1). Since p_k is correct, s is duly certified, formed and justified; from Lemma 1, $m \notin mistake_j[p_i]$; Thus, from line 18, Alg.1 and lines 15 and 27-31, Alg. 2, p_k is added to $extern\_susp_j[p_i][m.\text{id}]$ in line 31, Alg. 1 and the affirmation holds.
(2) If $|P| > 1$ (P has length greater than 1). We can assume by induction that the affirmation is true for the path $P - p_j$ between p_k and a correct process p_l, such that p_l and p_j are neighbors. For induction hypothesis, at some point in

time, p_k is added to $extern\_susp_l[p_i][m.\text{id}]$ and, afterwards, p_l send a message SUSPICION s (line 22, Alg. 1) with this information certified by p_k. Since channels are reliable, at some moment, p_j receives s (in line 17, Alg. 1). Since p_l is correct, s is duly certified, formed and justified; from Lemma 1, $m \notin mistake_j[p_i]$; thus, from line 18, Alg. 1 and lines 15 and 20-26, Alg. 2, p_k is added to $extern\_susp_j[p_i][m.\text{id}]$ in line 31, Alg. 1 and the affirmation holds.

From the above conditions, from $|KB_i^t| \geq 2f + 1$ and knowing that there is at most f faulty processes, it follows that, at some point in time, p_j executes line 33, Alg. 1 and, from lines 27-28, it adds $m.\text{id}$ 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 (it commits a security faulty). Then, at some point in time, every correct process $p_j \in \Pi$ will permanently include p_i in its $output_j$ set.*

Proof: A commission 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 (see Section 3). Notice that, due to the adoption of a broadcast communication pattern, mutant messages are not possible (see Section 4). Moreover, m is a certified message; otherwise, an undiagnosable faulty had been produced (see Section 3).

Since the network has Byzantine f-coverage, then there is a path P between p_i and each correct process p_j composed only by correct 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, at some point in time, p_j adds $\langle p_i, m\rangle$ to $byzantine_j$ and p_i to $output_j$.
(1) If $|P| = 1$ (P has length 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, Alg. 1. In both cases, the procedure ValidateReceived() (lines 13 and 18, Alg. 1) is invoked. This procedure will attest the non-validity of m at line 9, Alg. 2. For its turn, the procedure AddByzantine() (lines 1-3, Alg. 2) adds p_i to $output_j$ and $\langle p_i, m\rangle$ to $byzantine_j$, and the affirmation holds.
(2) If $|P| > 1$ (P has length greater than 1). We can assume by induction that the affirmation is true for the path $P - p_j$ between p_i and a correct 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, Alg. 1); since channels are reliable, at some moment, p_j receives s at line 17, Alg. 1. Since p_l is correct, s is duly certified, formed and justified; thus, as m is certified from line 18, Alg. 1 and lines 15 and 37-39, Alg. 2, 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 40-42, Alg. 1), p_i is definitely added to $output_j$ and the lemma follows. □

7.2 Eventual Weak Accuracy

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

Proof: Notice that the only invocation of AddByzantine() is in line 10, Alg. 2 into the procedure ValidateReceived(). From line 9, knowing that p_j is correct, 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 correct. 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, Alg. 1 and lines 33 and 37, Alg. 2), and the lemma follows. □

Lemma 5 *Let p_i be a correct process and m be a message required by $\mathcal{A}$. If every node in $range_i$ receives m from p_i in line 7, Alg. 1, then no correct process $p_j \in \Pi$ will invoke* AddInternalSusp (p_i, m).

Proof: The procedure AddInternalSusp() is called in 2 situations: (1): in the task T1, during the reception of messages from $\mathcal{A}$ (line 10); (2): in the procedure AddExternalSusp() (line 33), when the process receives more than f external suspicions regarding p_i.

Case 1: From the lemma hypothesis, nodes in $range_i$ receive m from p_i and then add p_i to their rec_from set in line 8, Alg.1. Thus, they do not invoke AddInternalSusp(p_i, m) in line 10. A process p_j out of $range_i$ cannot receive messages directly from p_i, thus, it will never add p_i to $known_j$ (lines 6-8, Alg. 2); thus, it will never invoke AddInternalSusp (p_i, m) in line 10. Both situations confirm Case 1.

Case 2: Notice that AddExternalSusp() is only invoked in lines 23 and 29 of Algorithm 2. Since a correct process only updates its $extern\_susp$ set on the execution of AddExternalSusp(), every external suspicion regarding a correct process was firstly generated as an internal suspicion (see lines 32 and 33, Alg. 1). From the same argument of Case (1), a correct 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 $extern\_susp_k[p_i][m.\text{id}]$, then a correct process will not adopt this suspicion since the authentication of the message is verified in line 21, Alg. 2. 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 faulty processes and the predicate in line 32, Alg. 2 is never satisfied. Thus, no correct process will invoke AddInternalSusp (p_i, m) in like 33, Alg. 1. The lemma, thus follows. □

Lemma 6 *Let p_i be a correct process. If there is a message m and a correct process p_j such that m.id $\in intern\_susp_j[p_i]$ during the run, then, at some point in the future, process p_j will invoke* AddMistake (p_i, m).

Proof: Two cases are possible.

Case 1: Process $p_j \in range_i$. Since p_i is correct, at some point, p_j receives m from p_i (duly certified, formed and justified) (line 17, Alg. 1). From the hypothesis of lemma, $m.\text{id} \in intern\_susp_j[p_i]$, thus, from lines 18, Alg. 1 and

lines 9 and 12, Alg. 2, since p_i is correct, p_j will call AddMistake (p_i, m) in line 13, Alg. 2.

CASE 2: Process $p_j \notin range_i$. By a similar argument used in Lemma 5, $p_i \notin known_j$. Thus, other correct 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 f-coverage, then there is a path P between p_k and p_j composed only by correct 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, at some point in time, each p_l in P invokes AddMistake (p_i, m), and thus $m \in mistake_l[p_i]$.
(1) If $|P| = 0$ (P has length 0). P has only p_k and for the same argument of Case (1), the affirmation holds.
(2) If $|P| > 0$ (P has length greater than 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, Alg. 1. Since p_l is correct, s is duly certified, formed and justified. Thus, for the execution of line 18, Alg. 1 and lines 19 and 32-36, Alg. 2, p_j calls ValidateReceived (p_i, m). Since p_i is correct, m is duly certified, formed and justified. Since m was forwarded by p_l, p_j calls AddMistake (p_i, m) in line 13, Alg. 2 (see lines 9 and 12-14) and the affirmation holds. The lemma thus follows. □

Lemma 7 *Let p_i be a correct process that satisfies $ByzRP$ (p_i). At some point in the future, every correct process $p_j \in \Pi$ is such that $p_i \notin output_j$.*

Proof: From Lemma 4, we can attest that p_i will never be added to $output_j$ in line 2, Alg. 2. From property $ByzRP$ (p_i), there exists a moment t after which every message m required by $\mathcal{A}$ in p_i is received by the neighbors of p_i in line 7, Alg. 1. 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 39, Alg. 1, at some point $intern\_susp_j[p_i] = \emptyset$. From Lemma 4, there is no pair $\langle p_i, - \rangle$ in $byzantine_j$; thus, p_i is removed from $output_j$ in line 4, Alg. 1 and the lemma holds. □

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

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

8 Conclusion and Future Work

This paper presented a Byzantine failure detector with two innovative characteristics: (i) it is suitable for dynamic distributed systems, i.e., systems with unknown composition, and (ii) it is asynchronous: it does not rely on timers

to detect progress failures. To enable the asynchronous failure detection, we conjecture to be necessary that the overlying algorithm is symmetrical, that is, that all nodes exchange messages at every step. One interesting conclusion is that communication through local broadcast, common in wireless networks, simplifies the security management, since the neighbors of a sender have an uniform view of the messages sent. Specifically, the protocol do not have to deal with *mutant messages*. As a future work, we plan to (i) extend the protocol with mobility management, (ii) implement the protocol for performance evaluation, and (iii) prove (or find a counterexample for) the impossibility of detecting Byzantine failures in an asynchronous manner with $1 \rightarrow n$ communication.

References

[1] Mostefaoui, A., Raynal, M., Travers, C., Patterson, S., Agrawal, D., El Abbadi, A.: From Static Distributed Systems to Dynamic Systems. In: Proc. of the 24th IEEE Symp. on Reliable Distributed Systems, Los Alamitos, CA, USA, IEEE Computer Society (2005) 109–118

[2] Lamport, L., Shostak, R., Pease, M.: The byzantine generals problem. ACM Trans. Program. Lang. Syst. **4**(3) (1982) 382–401

[3] Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. J. ACM **43**(2) (1996) 225–267

[4] Correia, M., Veríssimo, P., Neves, N.F.: Byzantine Consensus in Asynchronous Message-Passing Systems: a Survey. In: Resilience-building Technologies: State of Knowledge. Deliverable D12, Part Algo. RESIST Network of Excellence (September 2006)

[5] Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. J. ACM **32**(2) (1985) 374–382

[6] Larrea, M., Fernández, A., Arévalo, S.: Optimal implementation of the weakest failure detector for solving consensus. In: Proc. of the 19th IEEE Symp. on Reliable Distributed Systems, Washington, DC, USA, IEEE Computer Society (2000) 52–59

[7] Gupta, I., Chandra, T.D., Goldszmidt, G.S.: On scalable and efficient distributed failure detectors. In: Proc. of the 20th Annual ACM Symp. on Principles of Distributed Computing, New York, NY, USA, ACM Press (2001) 170–179

[8] Friedman, R., Tcharny, G.: Evaluating failure detection in mobile ad-hoc networks. Int. Journal of Wireless and Mobile Computing **1**(8) (2005)

[9] Mostefaoui, A., Mourgaya, E., Raynal, M.: Asynchronous Implementation of Failure Detectors. In: Proc. of the 2003 Int. Conf. on Dependable Systems and Networks, Los Alamitos, CA, USA, IEEE Computer Society (2003) 351

[10] Sens, P., Arantes, L., Bouillaguet, M., Simon, V., Greve, F.: An unreliable failure detector for unknown and mobile networks. In: Proc. of the 12th Int.

Conf. on Principles of Distributed Systems, Berlin, Heidelberg, Springer-Verlag (2008) 555–559

[11] Awerbuch, B., Holmer, D., Nita-Rotaru, C., Rubens, H.: An on-demand secure routing protocol resilient to byzantine failures. In: Proc. of the 1st ACM Workshop on Wireless Security, New York, NY, USA, ACM (2002) 21–30

[12] Aguilera, M.K., Chen, W., Toueg, S.: Heartbeat: A Timeout-Free Failure Detector for Quiescent Reliable Communication. In: Proc. of the 11th Int. Workshop on Distributed Algorithms, London, UK, Springer-Verlag (1997) 126–140

[13] Haeberlen, A., Kuznetsov, P., Druschel, P.: PeerReview: Practical accountability for distributed systems. In: Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07). (Oct 2007)

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

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

[16] Douceur, J.R.: The sybil attack. In: Revised Papers from the First Int. Workshop on Peer-to-Peer Systems, London, UK, Springer-Verlag (2002) 251–260

[17] Tang, K., Gerla, M.: MAC reliable broadcast in ad hoc networks. In: IEEE Military Communications Conference, 2001. MILCOM 2001. Communications for Network-Centric Operations: Creating the Information Force. Volume 2. (2001) 1008–1013

[18] Sun, M.T., Huang, L., Arora, A., Lai, T.H.: Reliable MAC Layer Multicast in IEEE 802.11 Wireless Networks. In: ICPP '02: Proceedings of the 2002 International Conference on Parallel Processing, Washington, DC, USA, IEEE Computer Society (2002) 527–536

[19] Guerraoui, R., Raynal, M.: The Information Structure of Indulgent Consensus. IEEE Trans. Comput. **53**(4) (2004) 453–466

[20] Fernández, A., Jiménez, E., Arévalo, S.: Minimal system conditions to implement unreliable failure detectors. In: 12th Pacific Rim Int. Symp. on Dependable Computing, Los Alamitos, CA, USA, IEEE Computer Society (2006) 63–72

[21] de Lima, M.S., Greve, F.G.P.: Protocolo Assíncrono para Detecção de Falhas Bizantinas em Sistemas Distribuídos Dinâmicos. In: Brazilian Symposium on Computer Networks and Distributed Systems, Recife, PE, Brazil, Brazilian Computing Society (May 2009) 887–900

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