

Failure, Disconnection and Partition Detection in Mobile Environment

Denis Conan — Pierre Sens — Luciana Arantes — Mathieu Bouillaguet

N° 6184

Mai 2007

Thème COM



*Rapport
de recherche*

Failure, Disconnection and Partition Detection in Mobile Environment

Denis Conan^{*}, Pierre Sens[†], Luciana Arantes[†], Mathieu Bouillaguet[†]

Thème COM — Systèmes communicants
Projet Regal

Rapport de recherche n° 6184 — Mai 2007 — 21 pages

Abstract: In mobile environment, nodes can move around and voluntarily leave or join the network. Furthermore, they can crash or be disconnected from the network due to the absence of network signals. Therefore, failure, disconnection and mobility may create partitions in wireless networks which should be detected for fault and disconnection tolerance reasons.

We present in this article an architecture of local and distributed detectors for mobile networks that detect failures, disconnections, and partitions. It is basically composed of three unreliable detectors: a heartbeat failure detector, a vector-based disconnection detector, and an eventually perfect partition detector. The latter ensures the convergence of detection information within a partition where all participants suspect the same sets of disconnected, unreachable, and faulty processes.

Key-words: partition detection, failure detector, mobile networks, distributed algorithms

^{*} GET / INT, CNRS Samovar, 9 rue Charles Fourier, 91011 Évry, France, Denis.Conan@int-edu.eu

[†] LIP6 — Université Paris 6 — INRIA, 4 Place Jussieu, 75252 Paris Cedex 05, France, Pierre.Sens,Luciana.Arantes,Mathieu.Bouillaguet@lip6.fr

Détection des fautes, des déconnexions et des partitions dans un environnement mobile

Résumé : Dans les environnements mobiles, les nœuds peuvent se déplacer en quittant et rejoignant le réseau. De plus, ils peuvent être sujets à des fautes ou des déconnexions dues à l'absence de signal. Ces déconnexions volontaires ou non peuvent créer des partitions qu'il est important de détecter.

Nous proposons dans cet article une architecture répartie permettant de détecter les fautes, les déconnexions et les partitions. Cette architecture est composée de trois détecteurs non fiables présents sur chacun des nœuds: un détecteur de fautes basé sur l'échange de messages de vie, un détecteur de déconnexions et un détecteur de partitions finalement parfait. Ce dernier assure pour chaque nœud correct la convergence des informations sur l'ensemble des nœuds non atteignables et fautifs.

Mots-clés : détection de partitions, détecteur de fautes, réseau mobile, algorithmes répartis

1 Introduction

Recent advancements in wireless data networking and portable information appliances have given rise to the concept of mobile computing. Users can access information and services irrespective of their movement and physical location. However, such an environment is extremely dynamic: Nodes can voluntarily disconnect themselves or move around; absence of wireless network signals can disconnect nodes from the network; nodes can fail and messages can be lost. Consequently, failure, disconnection, or mobility may cause a node or several of them to detach from the rest of the network, creating one or more network partitions.

As the geographic extent of the system grows or its connectivity weakens, network partitions tend to be more frequent. They may result in a reduction or degradation of services but not necessarily render the application completely unavailable. Partitions should keep working as autonomous distributed systems offering services to their clients as far as possible. Therefore, a mechanism for providing information to the application about network partition is highly important in wireless environments, and is the focus of this paper. We propose an eventually perfect unreliable partition detector for wireless systems. Similarly to an unreliable failure detector [5], an unreliable partition detector can be considered as a per process oracle, which periodically provides, for each process p , a list of processes suspected to be unreachable, that is those processes which are suspected of being in another partition than p 's one. A partition detector is unreliable in the sense that it can make mistakes. Two properties characterise a failure detector: *completeness* and *accuracy*. Roughly speaking, *completeness* sets requirements with respect to crashed processes, while *accuracy* restricts the number of false suspicions. By analogy, these two properties also characterise our partition detector, but with respect to reachable processes. Thus, our partition detector assures the following *completeness* and *accuracy* properties: A process p , which is correct, eventually detects every process that does not take part to p 's partition; and p eventually stops suspecting correct processes that belong to its partition.

Our partition detector is able to detect partitions due to disconnections as well as failures. The ultimate goal of characterising the nature of the partition is to help the decision-making process of applying countermeasures for fault tolerance and disconnection tolerance. Hence, in order to build our partition detector, a failure detector and a disconnection detector are required. Both detectors participate in our solution and the partition detector exploits information provided by them.

For detecting failures, we have chosen the class of heartbeat \mathcal{HB} failure detectors, proposed by [1, 2]. The reasons for such a choice are multiple. Firstly, \mathcal{HB} failure detectors can be used to achieve quiescent reliable communication, that is fair links that eventually stop sending messages, on top of asynchronous partitionable networks. They allow the conception of a quiescent stubborn broadcast primitive which both the disconnection detector and the partition detector of our solution need for diffusing information over the network. Another important feature of \mathcal{HB} failure detectors in our solution is that they do not output a list of suspected processes, but a vector of counters. Our partition detector makes use of such a vector for detecting network partitions: At each process p , the heartbeat sequence of a process which is not in the same partition of p 's one is bounded. Finally, the \mathcal{HB}

failure detector algorithm of [1, 2] also offers information about the topology of the network reachable through neighbours. We have then extended it in order to provide for each process p the information about the set of processes which are reachable from p through its own neighbours. This information is used by our partition detector.

In our approach, we consider that there is a local connectivity module at each mobile node which is responsible for informing whether that node can send messages or not [9]. It monitors resources such as energy power, memory space and wireless link quality by controlling one of their attributes such that, when the raw value of the attribute is below some threshold, the mobile node is disconnected. The objective of a connectivity module is to establish a connectivity mode (from strongly connected to disconnected) in a stabilised manner. However, such connectivity information needs to be spread over the network. Hence, when a node is locally notified of a disconnection, the disconnection detector that we propose will “try” to spread the disconnection information over the network, through its neighbours, by calling the broadcast primitive mentioned above.

The contribution of our paper is then threefold: (1) a modified version of the \mathcal{HB} failure detector of [1, 2] which besides offering information about failure suspicions and the possibility of building a stubborn reliable broadcast primitive, provides information about the reachability of nodes; (2) an unreliable disconnection detector that diffuses disconnection information through the network; and (3) an eventually perfect partition detector that, based on the information given by the two previous detectors, detects network partitions.

The remainder of this paper is organised as follows. In Section 2, we set out the distributed system model. Section 3 presents our global architecture and the basic primitives used throughout the paper. Section 4 describes the heartbeat failure detector for partitionable networks with terminal mobility and explains how the original algorithm was modified. The disconnection detector is presented in Section 5, and the partition detector in Section 6. We compare our contribution with related work in section 7 while section 8 concludes our work.

2 Distributed System Model

We consider a partially synchronous distributed system in which there are bounds on process speeds and on message transmission delays, these bounds are unknown, but they hold after some unknown time, which is called GST for Global Stabilisation Time [5]. The system consists of a set of n processes $\Pi = \{p_1, p_2, \dots, p_n\}$. The network of processes is a directed graph $G = (\Pi, \Lambda)$ where $\Lambda \subset \Pi \times \Pi$. Without lack of generality, we assume that there is one process per host. Process q is a neighbour of process p if and only if there is an unidirectional link from p to q . In mobile environments, a link that is not bidirectional means in practice that the two processes cannot rely on the same physical and logical resources in both directions. For argument’s sake, small devices like PDAs consume more power energy for emitting than for receiving messages on wireless networks, thus leading to non-uniform radio range. To simplify the presentation, we take the range \mathcal{T} of the clock’s

tick to be the set of natural numbers. Processes do not have access to \mathcal{T} : It is introduced for the convenience of the presentation.

Failure model Processes can fail by crashing, that is, by prematurely halting and then stop performing any further action for ever. We consider a network with two types of links: Links that are fair lossy and links that crash. A *fair lossy link* can intermittently drop messages and do so infinitely, but if a process p repeatedly sends a message m to process q , then q eventually receives m . Thus, if p sends m to q an infinite number of times and q is correct, then q receives m from p an infinite number of times. A *crashed link* stops forever transmitting messages. Following the terminology given in [1, 2], the network is said to be *partitionable*, that is a network in which some links may be unidirectional and may crash, in other words, the network may contain several unidirectional rings that intersect with each other. In addition, we assume that each correct process is connected to every other reachable process through a *fair path* that is a path containing only fair links and correct processes.

Disconnection model Processes can disconnect and reconnect. In connected mode, a process may send a message to its neighbours, while in disconnected mode, the resources of the process terminal are too low to send any application message but control messages may be transmitted for a while. We assume that every process ends its execution while being connected and does not crash while being disconnected¹. In the sequel, this translates into the assumption that a terminal that disconnects eventually reconnects. A moving node first disconnects from the network then it moves to a new location and finally reconnect to the network. We assume that mobile terminals eventually stop moving.

Partition model Since processes can fail and move, processes can be unreachable due to crashes or disconnections of other processes or links. We assume that the graph G is eventually strongly connected. In other words, every process ends its execution being not partitioned, that is partitions eventually merge into a global partition gathering all the correct processes. This assumption subsumes the two assumptions of eventually reconnection and eventually no more movement. In conclusion, the neighbourhood may change during execution due to hosts mobility and disconnections, and to crash and link failures.

3 General architecture and basic primitives

Figure 1 presents our global architecture. On each node, we provide a basic layer (\mathcal{BL}). The function of this layer is twofold. Firstly, it establishes a connectivity *mode* (from strongly connected to disconnected) in a stabilised manner. Secondly, it provides a list of current neighbours (*nghset*) by periodically calling of the networking layer. Each change in *mode* and *nghset* is notified to the upper layers.

¹In practice, the assumption means that the disconnected, and then terminating or faulty process does not succeed in leaving the set of participants Π . Then, a mechanism of leases at the application level will make the incriminated process leaving the set of participants.

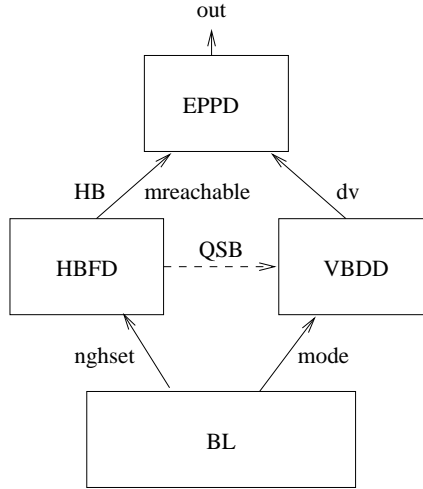


Figure 1: Overview of the software architecture of a node

On each node p , two detectors are plugged onto \mathcal{BL} : The heartbeat failure detector (\mathcal{HBFD}) and the vector-based disconnection detector (\mathcal{VBDD}). \mathcal{HBFD} outputs a per process vector of heartbeat counters HB and a set of mutually reachable processes $mreachable$. p and q are mutually reachable if there exists a fair path between p and q , and a fair path between q and p . Then, the p 's partition, denoted $partition(p)$, is the set of correct processes mutually reachable by process p . \mathcal{VBDD} outputs a per process disconnection counter dv . Disconnections and (re-)connections are numbered: Disconnection events are odd-numbered and reconnection events are even-numbered. Since processes are initially connected, the sequence number being 0, the first disconnection and reconnection are numbered 1 and 2, respectively, and so on.

At the upper level, the eventually perfect partition detector \mathcal{EPPD} uses information provided by both \mathcal{HBFD} and \mathcal{VBDD} to compute the set out that is the set of processes not in the same partition. Based on \mathcal{HBFD} , we provide a *quiescent stubborn broadcast* primitive QSB used by \mathcal{VBDD} to diffuse disconnection and reconnection information.

Each process can use the following primitives to communicate:

- $send(dest, m)/receive(from, m)$: Two basic point-to-point communication functions to send (*resp.* receive) message m to (*resp.* from) its neighbour $dest$ (*resp.* $from$). When a message is issued from a local component, $local_send(dest, m)$ and $local_receive(from, m)$ functions are used where $from$ and $dest$ are the name of the component (\mathcal{HBFD} , \mathcal{VBDD} or \mathcal{BL}).
- $broadcast(m)$: This function diffuses message m over fair links to all correct processes by implementing QSB . This primitive provides the abstraction of stubborn links

hiding the retransmission mechanisms used to make somewhat reliable the transmission of messages. A formulation of the stubborn delivery property is as follows [7]: If the sender p , which does not crash, sends a message m to q that is correct, and p is able to indefinitely delay the sending of any further message, then q eventually receives m . An important practical consideration is that stubborn links require only a bounded buffer space (minimum of one message). The quiescence property ensures that only a finite number of messages is sent when broadcast is invoked a finite number of times. *QSB* uses *HBFD*. Due to the lack of space, we do not present the broadcast algorithm in this paper.

HBFD, *VBDD* and *EPDD* are characterised by both completeness and accuracy properties defined as follows:

- *HB-Completeness*: At each correct process p , the heartbeat counter of every process not in $partition(p)$ is bounded.
- *HB-Accuracy*: At each correct process p , the heartbeat counter of every process is non-decreasing. The heartbeat counter of every process in the partition of p is unbounded.
- *VBDD-Completeness*: Eventually all disconnections and reconnections of correct processes are seen by every correct process.
- *VBDD-accuracy*: No process sees a disconnection (*resp.* reconnection) before the disconnection (*resp.* reconnection) effectively occurs.
- *EPDD-Completeness (Strong partition completeness)*: If some process q remains unreachable from a correct process p , then eventually p will always suspect q of not belonging to $partition(p)$.
- *EPDD-Accuracy (Eventual strong partition accuracy)*: If some process q remains reachable from a correct process p , then eventually p will no longer suspect q of not belonging to $partition(p)$.

4 Failure detection

Our failure detector *HBFD* is based on the class of heartbeat failure detectors, proposed by [1, 2]. Such a choice is firstly explained by the need to build quiescent algorithms, that is algorithms that eventually stop sending messages in partitionable networks. In [2], the authors prove that, with fair links and in the presence of process crashes, quiescent reliable communication are impossible without having a failure detection which provides an output with bounded size. Hence, they propose in the paper the class of heartbeat failure detectors which can be used to circumvent this impossibility result. Another reason that justifies our choice is that heartbeat failure detectors are not time-out-based unlike the majority of current failure detector implementations that both provide lists of suspected processes and increase time-outs in case of false suspicions.

Heartbeat failure detectors provide for each process p a vector of counters $HB = [n_1, n_2, \dots, n_k]$ where each n_j is a positive integer corresponding to the number of heartbeats received by process p from p_j . Thus, n_j is the “heartbeat value of p_j at p ”. Intuitively, n_j increases as long as p_j is correct, not disconnected, and reachable from p . Notice that heartbeat failure detectors provide the vector HB without any treatment or interpretation. Then, other detectors, as our partition detector \mathcal{EPPD} , can periodically obtain the current value of HB vector from \mathcal{HBFD} in order to deduce lists of suspected processes.

Beside the heartbeat vector HB , our failure detector \mathcal{HBFD} gives information about the topology of the network since each process keeps information about which processes can be reachable through its neighbours. For each neighbour r of process p , \mathcal{HBFD} builds the set of processes mutually reachable from p through r . This set is called the *reachability* set of p through r and the vector *mreachable* gathers the set of *reachability* sets of all the neighbours of p . The property of mutual reachability, can be expressed as follows: At each correct process p , for each neighbour r , the *reachability* set for r (*mreachable*[r]) eventually contains all the correct processes (*e.g.*, q), such that there is a simple fair path from p to q through r and a simple fair path from q to p . In a simple fair path, no process appears more than once. Furthermore, \mathcal{HBFD} can also accept requests for emptying some of the reachable sets in order to restart an accumulation phase of topology discovery. This functionality is used by our partition detection \mathcal{EPPD} , described in Section 6, when a failure or a disconnection is detected.

\mathcal{HBFD} which runs on each node p is presented in Algorithm 1. It is based on the algorithm for partitionable networks described in [1]. The changes we have made are related to the addition of nodes mobility, discovery of the network topology through neighbours, and the reduction of the number of heartbeat messages sent over the network. The variables HB and *mreacheable* respectively store the per process heartbeat counters and the per process mutual *reachability* sets, as previously described. The set *nghbrs* controls the current neighbours of p , while the set *paths* gathers all the paths of which p is aware since its last heartbeat sending. Algorithm 1 is executed by process p ($p \in \Pi$), and it is divided into five parallel tasks. It provides to the upper client, *e.g.* the partition detector \mathcal{EPPD} , the heartbeat vector HB and the reachable sets (sets of *mreacheable*) (line 17). The principle of the algorithm is the piggy-backing of fair paths in heartbeat messages.

The first task (lines 1–5) corresponds to the code block executed at the creation of the heartbeat failure detector. The second task (lines 6–9) is triggered when the neighbourhood changes. Such an information is provided by \mathcal{BL} (*cf.* section 3). This task controls the mobility of nodes and therefore the current set *nghbrs* of neighbours of p (line 9). Furthermore, the entries of *mreacheable* corresponding to those processes that are no longer neighbours of p are set to empty (line 7) since they cannot be reached anymore from p through old neighbours. However, new neighbours of p are seen as reachable (line 8).

In the third task (lines 10–17), process p periodically increments its own heartbeat and adds itself to *paths*, which already contains all paths received in heartbeat messages during the last period of time. However, before sending to all its neighbours a new heartbeat message which includes such a variable (line 15), p verifies in line 14 if its previous heartbeat

Algorithm 1: Heartbeat Failure Detector \mathcal{HBFD}

```

1  upon initialisation do
2     $ngnbrs \leftarrow \emptyset$  {neighbourhood at  $p$ }
3     $HB[1..|\Pi|] \leftarrow \{0, \dots, 0\}$  {heartbeat vector at  $p$ }
4     $mreachable[1..|\Pi|] \leftarrow \{\emptyset, \dots, \emptyset\}$  {processes reachable through neighbours from  $p$ }
5     $paths \leftarrow \emptyset$  {set of paths received in heartbeats during last period of time}
6  upon local_receive( $\mathcal{BL}, nghset$ ) do
7    for all  $q \in nghbrs \setminus nghset$  do  $mreachable[q] \leftarrow \emptyset$ 
8    for all  $q \in nghset \setminus nghbrs$  do  $mreachable[q] \leftarrow \{q\}$ 
9     $ngnbrs \leftarrow nghset$ 
10 periodically do
11    $HB[p] \leftarrow HB[p] + 1$ 
12    $paths \leftarrow paths \cup \{\{p\}\}$ 
13   for all  $q \in nghbrs$  do
14     for all  $path \in paths$  :  $(\exists r \in path : r \text{ appears more than twice in } path)$  do  $paths \leftarrow$   

15      $paths \setminus path$ 
16     for all  $q \in nghbrs$  do  $send(q, \langle \mathcal{HBFD}, paths \rangle)$ 
17      $paths \leftarrow \emptyset$ 
18     local_send( $\mathcal{EPPD}, \langle \mathcal{HBFD}, HB, mreachable \rangle$ )
19   upon receive( $q, \langle \mathcal{HBFD}, paths \rangle$ ) do
20     for all  $path \in paths$  do
21       for all  $r \in \Pi$  :  $r$  appears after  $p$  in  $path$  do  $HB[r] \leftarrow HB[r] + 1$ 
22       if  $\exists r \in \Pi$  :  $r$  appears right next to  $p$  in  $path$  then
23         for all  $s \in \Pi$  :  $s$  appears after  $r$  in  $path$  do  $mreachable[r] \leftarrow mreachable[r] \cup \{s\}$ 
24         endif
25        $paths \leftarrow paths \cup \{(path \cdot p)\}$ 
26     endfor
27   upon local_receive( $\mathcal{EPPD}, \langle \mathcal{EPPD}, procset \rangle$ ) do
28     for all  $s \in procset$  do  $mreachable[s] \leftarrow \emptyset$ 

```

messages have not already completed two cycles. In this case, such a path will be removed from $paths$ (line 14). As we will show in an example below, some topologies require that heartbeat messages completing a first cycle need to be forwarded to the initiators so that those processes add processes of the cycle to their mutually reachable sets. Instead of the condition “more than twice”, one can devise a solution forwarding these heartbeat messages only to the initiators and removing them from $paths$ for the other neighbours (such that the condition becomes “more than once” like in [1, 2]). At the end of the third task, the heartbeat failure detector notifies its clients, \mathcal{EPPD} in our case, about new updated information concerning the heartbeat vectors and reachability sets (line 17).

The fourth task (lines 18–25) handles the reception of messages by p of the form $\langle \text{HBFD}, paths \rangle$. Upon receiving it from process q , for each $path \in paths$ with $path = (p_1 \cdot \dots \cdot p_i \cdot \mathbf{p} \cdot r \cdot p_j \cdot \dots \cdot p_k \cdot q)$, p adds the processes $(p_j \cdot \dots \cdot p_k \cdot q)$, which appears after its neighbour r , to $mreachable[r]$ (lines 21–22). Therefore, $mreachable[r]$ contains a list of processes that can be mutually reached from p through r . In addition, process p increases the heartbeat counters of all the processes that appear after p in $path$, that is all the processes of the sequence $(r \cdot p_j \cdot \dots \cdot p_k \cdot q)$ (line 20), since they are not suspected by p . Process p appends then itself to $path$ and stores the new path in $paths$ (line 24). Notice that in this case, p is also reachable from $(p_j \cdot \dots \cdot p_k \cdot q)$ through their respective neighbours.

Finally, the fifth task (lines 26–27) empties some entries of $mreachable$. As previously explained, this functionality is used by the partition detector \mathcal{EPPD} , described in section 6.

Example of execution of HBFD

In order to explain how nodes dynamically discover which are the other nodes reachable through their respective neighbours, we show in Figure 2 the scenario of an execution of Algorithm 1, considering a topology with five nodes.

Node 1 starts by sending to its neighbour node 2 a heartbeat message that contains the variable $paths_1$ which, in this case, includes just itself, as shown in Figure 2.(a) (line 12). Upon receiving it (cf. Figure 2.(b)), node 2 appends itself to all the received paths, adding the latter to its variable $paths_2$ (line 24). Notice that it does not update its variable $mreachable_2$ since it is not included in any of the received paths. Next, the set $\{2\}$ is being added to $paths_2$ (line 12) and a new heartbeat message is sent to its neighbours. Both nodes 1 and 3, outgoing neighbours of 2, receive it.

In figure 2.(c), both nodes 1 and 3 receive the above heartbeat message, while in Figures 2.(d) and 2.(e), node 4 receives the heartbeat messages sent by node 3, and node 5 receives the heartbeat messages sent by node 4, respectively. Next, when node 2 receives the heartbeat message from node 5 (cf. Figure 2.(f)), it finds itself in some of the received paths. Therefore, line 22 of the algorithm is executed and node 2 adds nodes 4 and 5 to $mreachable_2[3]$, that is, these nodes are mutually reachable from node 2 through its neighbour node 3.

Finally, Figure 2.(g) shows that the content of variable $paths_1$ of node 1 after having received the second heartbeat message from node 2. In the scenario, we consider that node 1 has not sent any new heartbeat message after the reception of the first heartbeat message

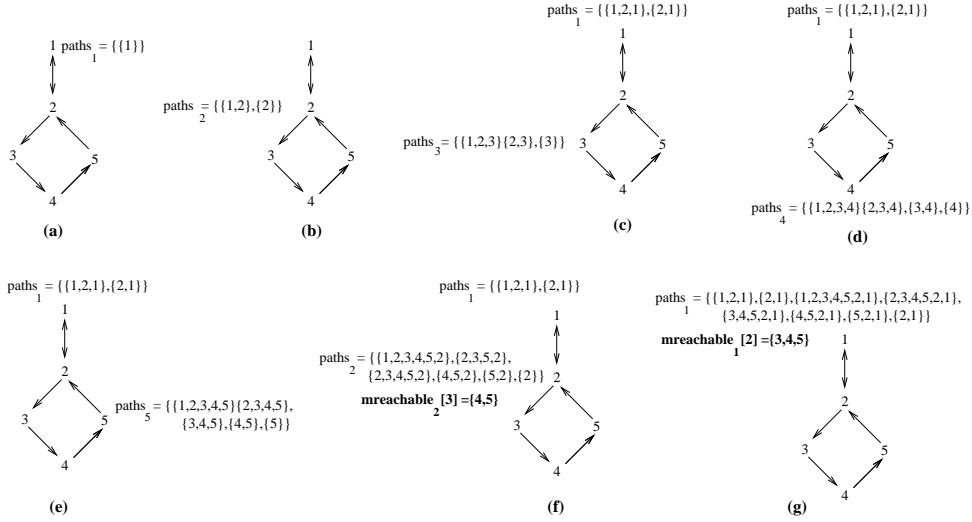


Figure 2: Example of reachability set dynamic construction

from node 2. Node 1 then updates its variable *mreachable* ($mreachable_1[2] = \{3, 4, 5\}$) since these nodes appear after its neighbour 2 in path $\{1, 2, 3, 4, 5, 2, 1\}$.

Sketch of proof \mathcal{HB} -completeness: The proof is by contradiction. Let q be a process that is not in the partition of p ($p \neq q$). Assumes that $HB[q]$ is not bounded. Then, p receives an infinite number of times messages $\langle \text{HBFD}, paths \rangle$, where there exists a path P in $paths$ which contains q after p . This path is of the form $P = (p_1 \dots p \dots q \dots p_k)$. Since p receives an infinite number of messages from p_k , the link $p_k \rightarrow p$ is fair. By repeated application, for each $j = k - 1, \dots, 1$, the link $p_j \rightarrow p_{j+1}$ is fair. Thus, in P , $(p \dots q)$ is a fair path from p to q and $(q \dots p_k \cdot p)$ is a fair path from q to p . Therefore, p and q are in the same partition —a contradiction.

\mathcal{HB} -accuracy: The first part (the heartbeat counter of every process is nondecreasing) is obvious since $HB[q]$ can only be changed in lines 11 and 20. For the second part (the heartbeat counter of every process in the partition of p is unbounded), two cases are possible. Let q be a process in the partition of p . If $q = p$, then line 11 is executed infinitely often (since p is correct) and $HB[p]$ at p is unbounded. Now, assume $q \neq p$ and let $(p_1 \dots p_i)$ be a simple fair path from p to q , and $(p_i \dots p_k)$ be a simple fair path from q to p , so that $p_1 = p_k = p$ and $p_i = q$. For $j = 1, \dots, k - 1$, let $P_j = (p_1 \dots p_j)$. By induction on j , we can show that, for each $j = 1, \dots, k - 1$, p_j sends messages $\langle \text{HBFD}, paths \rangle$ to p_{j+1} an infinite number of times, where there is a path P in $paths$ such that $P = subpath \cdot P_j$. For $j = k - 1$, this claim shows that a neighbour of p sends messages $M = \langle \text{HBFD}, paths \rangle$ to p an infinite number of times, where there is a path P' in $paths$ such that $P' = subpath \cdot P_{k-1}$.

Since p is correct and by the fairness property of the links, p receives messages of the form of M an infinite number of times. Since q appears after p in P_{k-1} , $HB[q]$ is incremented an infinite number of times (line 20). Therefore, $HB[q]$ is unbounded.

5 Disconnection detection

The connectivity information provided by \mathcal{BL} (cf. Section 3) remains local to the mobile node. In addition, a mobile node can voluntarily decide to disconnect itself from the network. Hence, as we want in our approach to make the difference between a disconnection and a failure, the disconnection/reconnection information of nodes should be spread over the network.

We consider that when a node is disconnected from the network, it does not send application messages anymore. However, this does not mean that control messages sent by fair links cannot be transmitted; in other words, physical transmission may be still possible for a while. Thus, contrary to failures which are unexpected, there is a lapse of time between the connectivity detection of the node “disconnected” and the effective physical disconnection. Such a lapse of time can then be used for alerting remote processes of a node disconnection. Clearly, in the case of a sudden disconnection, no disconnection message can be sent and the disconnection will be detected as a failure by the failure detector that runs on correct and connected processes. This false suspicion will last for the duration of the disconnection and will be corrected when the disconnected process reconnects. On the other hand, in the case where the end-user disconnects themselves voluntarily, we consider that the middleware service responsible for isolating the user’s node waits for a short while before actually performing the disconnection, thus allowing the transmission of control messages before the interruption of communication. In other words, since it has been initiated by the end-user, the disconnection information may be sent to other processes before actually isolating the node.

Then, we introduce the concept of unreliable vector-based disconnection detector, \mathcal{VBDD} , similar to the one of unreliable failure detection. When a process is notified of a disconnection either by \mathcal{BL} or voluntarily by the end-user, \mathcal{VBDD} “tries” to transmit the disconnection information to all the processes by calling \mathcal{QSB} . The same principle is used both to diffuse disconnection and reconnection information.

By analogy with heartbeat failure detectors, the disconnection detector does not output a list of disconnected processes, but provides a per process vector, named dv , of disconnection/reconnection event counters. If $dv[q]$ of process p contains an even value, q is considered to be seen as connected by p , otherwise it is considered to be disconnected. Notice that such an interpretation of the disconnection vector’s entries is done afterwards by the partition detector \mathcal{EPPD} . It is worth mentioning that \mathcal{VBDD} considers only disconnection/reconnection of correct processes. Indeed, by construction, the disconnection detector is not able to suspect processes of being faulty. So, as mentioned in Section 2, we assume that every process does not crash while being disconnected.

The algorithm for process p of our disconnection detector \mathcal{VBDD} for partitionable networks which supports node mobility is presented in Algorithm 2. It has four tasks. The principle of the algorithm is to diffuse via \mathcal{QSB} the disconnection vector dv into VBDD messages when one of the following events is triggered: new neighbourhood, voluntary disconnection, connectivity mode change, or delivery of a VBDD message with new information. The local vector dv keeps information about process disconnection/reconnection, as previously described. The local variable *voluntaryDisc* indicates whether the end-user has requested a voluntary disconnection, and *mode* is a variable which is updated with the information provided by \mathcal{BC} about the connectivity of node p itself, the latter information being inferred from raw data from the execution context. One of the role of VBDD is to infer a logical connectivity mode by mixing these two inputs and storing the result in $dv[p]$, the other role being to build a coherent distributed view of disconnection events.

The first task (lines 1–4) corresponds to the code block executed at the creation of the disconnection detector. Every process is considered to be connected at the beginning of the execution (lines 2–4). The next task (lines 5–10) allows the end-user to voluntarily disconnect or reconnect (by opposition to involuntary disconnections or reconnections detected by \mathcal{BC}). The assignment of the variable *voluntaryDisc* (line 6) is followed by the propagation of this new disconnection event to every neighbour (line 9). Naturally, voluntary disconnections outdo involuntary disconnections/reconnections. Thus, when p is not already disconnected, either voluntarily or involuntarily ($mode \neq 'd'$), a voluntary disconnection effectively disconnects the process. Similarly, when p is currently voluntarily disconnected and involuntarily connected ($mode \neq 'd'$), a voluntary reconnection effectively reconnects the process. The third task (lines 12–18) is executed when there is a change in the connectivity mode which is detected by \mathcal{BC} . If the node becomes disconnected or connected, and the end-user did not ask for a voluntary disconnection (condition of the *if* at line 13), p broadcasts the new disconnection event (line 15).

The last task (lines 19–24) is responsible for the updating of the disconnection vector as a result of the delivery of a newly-received disconnection vector (dvq), contained in a VBDD message. At line 20, dv is compared with dvq . If one or more of the values of dv entries are smaller than the dvq 's ones, dv is updated with the maximum of the entries of the two vectors (line 21) and dv is broadcast (line 22). This new disconnection message is going to update the disconnection vector of other processes that might not be aware of some disconnection/reconnection events. At p , \mathcal{VBDD} then provides to the upper detector \mathcal{EPPD} , which runs on p , the disconnection vector dv (line 23).

Sketch of proof \mathcal{VBDD} -completeness: In the following, the generic expression “disconnection event” is used to refer to all VBDD messages. There are four possible cases: (1) p successfully sends a VBDD message to all its neighbours; (2) p is physically disconnected just before sending a VBDD message.; (3) p successfully sends the disconnection event to at least one correct and connected process q ; and (4) p moves. In the first case, by the eventually strongly connected hypothesis (section 2 page 5) of the distributed model and by the stubborn delivery property (section 3 page 6), all the correct processes eventually

Algorithm 2: Vector-based disconnection detector \mathcal{VBDD}

```

1  upon initialisation do
2     $dv[1..|\Pi|] \leftarrow \{0, \dots, 0\}$            {Vector of disconnection sequence numbers at  $p$ }
3     $voluntaryDisc \leftarrow false$              {true if voluntary disconnection of  $p$ }
4     $mode \leftarrow 'c'$                        {connectivity mode at  $p$ }
5  upon voluntary disconnection/reconnection by the end user do
6     $voluntaryDisc = \neg voluntaryDisc$ 
7    if  $mode \neq 'd'$  then
8       $dv[p] \leftarrow dv[p] + 1$ 
9      broadcast( $\langle \mathcal{VBDD}, dv \rangle$ )
10     local_send( $\mathcal{EPPD}, \langle \mathcal{VBDD}, dv \rangle$ )
11   endif
12  upon local_receive( $\mathcal{BL}, newMode$ ) do
13   if  $newMode \neq mode \wedge \neg voluntaryDisc$  then
14      $dv[p] \leftarrow dv[p] + 1$ 
15     broadcast( $\langle \mathcal{VBDD}, dv \rangle$ )
16     local_send( $\mathcal{EPPD}, \langle \mathcal{VBDD}, dv \rangle$ )
17   endif
18    $mode \leftarrow newMode$ 
19  upon receive( $q, \langle \mathcal{VBDD}, dvq \rangle$ ) do
20   if  $\neg(\forall r \in \Pi : dv[r] \geq dvq[r])$  then           { $\neg(dv \geq dvq)$ }
21     for all  $r \in \Pi$  do  $dv[r] \leftarrow \max(dv[r], dvq[r])$ 
22     broadcast( $\langle \mathcal{VBDD}, dv \rangle$ )
23     local_send( $\mathcal{EPPD}, \langle \mathcal{VBDD}, dv \rangle$ )
24   endif

```


deliver a VBDD message containing a disconnection vector greater than or equal to dv . In the second case, the VBDD message is sent whenever p reconnects. This is because of the properties of the stubborn primitive: messages are saved in the mobile terminal's buffer and are sent when the terminal reconnects. p then successfully disseminates this message or a newer one as done in the first case. In the third case, if q is not physically disconnected after delivering the disconnection event of p , then it successfully disseminates this event as done in the first case. But, if q is physically disconnected right after the delivery of the VBDD message of p , again, the disconnection event or a newer one is disseminated whenever process p or q reconnects to the network, as in the second case. Finally, if q successfully transmits the disconnection event to at least one of its neighbours, this is again the third case by recursion. Clearly, by the eventually strongly connected hypothesis and by the stubborn delivery property, a VBDD message containing dv or a greater dv is eventually delivered by all the correct processes. In the last case, neighbourhood changes provoke the diffusion of a VBDD message to the new neighbourhood. Clearly, the decomposition into the first three cases just studied before is also valid, leading to the same conclusion.

VBDD-accuracy: First of all, notice that the p th entry of the disconnection vector is only incremented at process p when p executes the code statements corresponding to voluntary disconnections/reconnections (line 5) or to involuntary disconnections/reconnections (line 12). Next, other processes update the p th entry of their disconnection vector only when treating VBDD messages. Therefore, the p th entry of the disconnection vector of process q ($q \neq p$) is always less than or equal to the p th entry of the disconnection vector of p .

6 Partition Detection

A network can become partitioned due to link or node failures as well as node disconnections. In other words, some nodes may become unreachable to others. Thus, in this section, we present our generic partition detector \mathcal{EPPD} , which establishes, for process p , the set of suspected processes which are not in $partition(p)$. To this end, \mathcal{EPPD} exploits information given by both \mathcal{Hbfd} and \mathcal{VBDD} .

\mathcal{EPPD} has been defined based on the completeness and accuracy properties as described in section 3. Its specification is inspired from [3], where a failure detection for partitionable group systems is presented. The authors formalise the stability conditions that are necessary for solving group membership in asynchronous systems. The specification is close to our approach because it is expressed by the reachability between pairs of processes rather than on individual processes being correct or crashed. In our work, we prefer the term partition detector to underline the fact that the specification is partition-oriented. Considering process p , \mathcal{EPPD} suspects those processes that do not belong to the same partition of p . However, it provides a list of suspected processes only for stabilised periods. Thus, even if we consider that the network graph is eventually strongly connected (see section 2) and that client algorithms eventually terminate, the objective of \mathcal{EPPD} specification is to allow algorithms, adapted for partitionable networks, to terminate their execution with a smaller number of processes during stabilised periods.

Algorithm 3 describes our partition detector \mathcal{EPPD} . It tries to discover network partitions using both the heartbeat vector and reachability information provided by \mathcal{HBFDD} , as well as the disconnection vector provided by \mathcal{VBDD} . All local variables are initialised in the first task (lines 1–5). The set of suspected processes that do not belong to the same partition of p is noted as *out* (for “out” of the partition). Hence, in order to be able to provide such set, we have introduced in the algorithm of \mathcal{EPPD} a time-out. It allows to build an eventually perfect detector based on heartbeat vector values given by \mathcal{HBFDD} (see variables HB and $prevHB$). At the same time, the reachability information is stored in variable $mreach$. \mathcal{EPPD} also parses disconnection vectors provided by \mathcal{VBDD} (see variable dv).

The second task of \mathcal{EPPD} (lines 6–13) monitors the disconnection events received from the vector-based disconnection detector \mathcal{VBDD} . Each entry of the disconnection vector is analysed. When \mathcal{VBDD} at process p notifies a new disconnection of process q , all processes considered not to be reachable anymore from p due to the disconnection of q , are added to *out* by the call of the procedure **add** (lines 26–33). The latter procedure parses the reachability sets to detect which are the processes that became unreachable from p , that is processes that appear to be reachable through no other neighbour. Then, in order to forget those processes that were previously reachable from p through q (when q was a neighbour of p), the reachable set $mreach[q]$ is reset. This is done by sending a message to the local \mathcal{HBFDD} (line 11). On the other hand, when the event notified by \mathcal{VBDD} is a new reconnection of q , that is q is no more suspected to be unreachable from p , q is removed from *out* (procedure **remove** at lines 34–35).

The last task (lines 14–25) monitors failure by examining both the heartbeat counters and reachability sets provided by \mathcal{HBFDD} . For each process in Π , the difference between the values of the new heartbeat counter HB and the old heartbeat counter $prevHB$ is compared against the failure detection threshold value 1, which is assumed to be strictly greater than 0. In the test of line 17, a failure suspicion of process q detected at process p is new if (1) p suspects q ($HB[q] - prevHB[q] < 1$), and (2) q is not already suspected ($q \notin out$). In this case, process q and all the processes that became unreachable from p due to the failure suspicion of q are added to *out* by the call of the procedure **add**. Contrariwise, a false suspicion of process q detected at process p (line 20) is new if (1) p does not suspect q ($HB[q] - prevHB[q] \geq 1$), (2) q is already suspected ($q \in out$), and (3) p is not seen as disconnected. In this case, q is removed from *out* by the call to the procedure **remove** (lines 34–35).

Sketch of proof Strong partition completeness: There are 3 cases to consider: q is included in *out* (1) either due to the disconnection detection of q , (2) or due to the failure suspicion of q , (3) or even due to the partition suspicion of q following the disconnection or the failure of another process r . In the first case, from the \mathcal{VBDD} -completeness property, we know that the corresponding disconnection event of q is eventually delivered by every correct process. Thus, all the correct processes in $partition(q)$ execute the code block at lines 6–13, and q is added to the set *out*. In the second case, from the \mathcal{HB} -completeness property, we

Algorithm 3: Eventually Perfect Partition Detector \mathcal{EPPD}

```

1  upon initialisation do
2     $out \leftarrow \emptyset$                                 {processes suspected to be out of  $partition(p)$ }
3     $mreach \leftarrow \{\emptyset, \dots, \emptyset\}$       {mutually reachable sets}
4     $prevHB[1..|\Pi|] \leftarrow \{0, \dots, 0\}$         {previous heartbeat counters}
5     $prevDV[1..|\Pi|] \leftarrow \{0, \dots, 0\}$         {previous disconnection vector}
6  upon local_receive( $\mathcal{VBDD}, \langle \mathcal{VBDD}, dv \rangle$ ) do
7    for all  $q \in \Pi : prevDV[q] < dv[q]$  do
8      if  $dv[q] \bmod 2 = 1$  then                       {new disconnection}
9        call add( $q$ )
10     else call remove( $q$ )                             {new reconnection}
11     local_send( $\mathcal{HBFD}, \langle \mathcal{EPPD}, out \rangle$ )
12   endfor
13    $prevDV \leftarrow dv$ 
14  upon local_receive( $\mathcal{HBFD}, \langle \mathcal{HBFD}, HB, mreachable \rangle$ ) do
15      $mreach \leftarrow mreachable$                     {new mutually reachable sets}
16   for all  $q \in \Pi : q \neq p$  do
17     if  $HB[q] - prevHB[q] < 1 \wedge q \notin out$  then   {new failure suspicion}
18       call add( $q$ )
19       local_send( $\mathcal{HBFD}, \langle \mathcal{EPPD}, out \rangle$ )
20     else if  $HB[q] - prevHB[q] \geq 1 \wedge q \in out \wedge dv[p] \bmod 2 = 0$  then   {false suspicion}
21       call remove( $q$ )
22       local_send( $\mathcal{HBFD}, \langle \mathcal{EPPD}, out \rangle$ )
23     endelseif
24   endfor
25    $prevHB \leftarrow HB$ 
26  procedure add( $q$ )
27     $out \leftarrow out \cup \{q\}$ 
28    if  $q = p$  then  $out \leftarrow \Pi \setminus \{p\}$       {local disconnection}
29    else                                             {disconnection or failure of a remote process}
30      for all  $s \in \Pi : s \in mreach[q] \wedge s \neq q \wedge dv[s] \bmod 2 = 0$  do
31        if  $\nexists u \in \Pi : u \neq q \wedge dv[u] \bmod 2 = 0 \wedge s \in mreach[u]$  then  $out \leftarrow out \cup \{s\}$ 
32      endfor
33    endelse
34  procedure remove( $q$ )
35     $out \leftarrow out \setminus \{q\}$ 

```

know that the heartbeat counter of every process $q \notin \text{partition}(p)$ is eventually bounded. Thus, for all the processes in $\text{partition}(p)$, the condition at line 17 is eventually true and q is added to the set *out*. The third case, the partition suspicion of q due to the disconnection detection or the failure suspicion of another process r , is included in the previous cases. Process q is added to the set *out* by the call of the procedure `add(r)` at lines 29–31.

Eventual strong partition accuracy: The proof is by contradiction. Assume a correct process $q \in \text{partition}(p)$ is permanently suspected by p , that is $q \in \text{out}$. From the \mathcal{HB} -accuracy property, we know that the heartbeat counter of every process in $\text{partition}(p)$ is unbounded. In addition, from the \mathcal{VBDD} -accuracy property and since $q \in \text{partition}(p)$, q is connected. Therefore, the condition of the test at line 20 is eventually true and q is eventually removed from the set *out* —a contradiction.

6.1 Making the difference between disconnection and failure

In the partition detector \mathcal{EPPD} , the disconnection events and the reachability information diminish the detection latency. For instance, if a process q is seen as disconnected by \mathcal{VBDD} before being detected as faulty by \mathcal{HBFDD} , \mathcal{EPPD} immediately declares q out of $\text{partition}(p)$. Similarly, consider a process r reachable from p only through the neighbour q ; as soon as q becomes unreachable, p declares r out of $\text{partition}(p)$, without waiting for the detection by \mathcal{HBFDD} . In other words, \mathcal{EPPD} builds at process p the set of processes suspected to be unreachable from p but without providing any hint about the cause of the partition.

We have designed a more elaborated version of partition detection which adds semantical rules to interpret the detection events from lower-layers and which constructs, over the correct processes of $\text{partition}(p)$, a convergence on the following three sets: (1) the set of processes suspected to be faulty by all the processes in $\text{partition}(p)$, (2) the set of processes seen as disconnected by all the processes in $\text{partition}(p)$, and (3) the set of processes suspected, by all the processes in $\text{partition}(p)$, to be partitioned due to the failure or the disconnection of another process. Note that the union of these sets is equal to the set *out* of \mathcal{EPPD} . In this version of the partition detector, the objective is not the accuracy of the individual sets, but the convergence on the content of each set. The ultimate goal of characterising the cause of the partition is to help the decision-making process of applying countermeasures for fault tolerance (*e.g.*, replace faulty processes of a replicated state machine) and disconnection tolerance (*e.g.*, open new links or adapt applications' behaviour).

Contrariwise to Algorithm 3, the new algorithm is a distributed algorithm in which each process diffuses, in partition messages `PART` and using \mathcal{QSB} , its estimate of the knowledge of which processes are faulty, which ones are disconnected, and which ones are partitioned. In addition, in order to converge to a common knowledge in a partition, the algorithm applies semantical rules which prioritise detection hints. The intuitive ideas behind the scene are (1) a disconnection is “stronger” than a partition, which is “stronger” than a failure, and (2) prefer hints from local detectors to hints from remote ones. Of course, one can design other semantical rules than these ones.

The first intuitive idea states that whenever possible, be optimistic, and prefer considering a process being disconnected rather than suspecting it to be partitioned, rather than

suspecting its failure. For instance, a failure suspicion is new when the corresponding process is not already suspected to be faulty, but also neither seen disconnected nor suspected to be partitioned. According to the second intuitive idea, note that the information provided by the local instance of \mathcal{VBDD} may contradict the information contained by PART messages. The new algorithm gives priority to the hints provided by \mathcal{VBDD} . The reason is that the \mathcal{VBDD} -completeness stipulates that eventually all disconnections and reconnections are seen by all the correct processes in the same partition: Even if \mathcal{VBDD} can temporarily provides false hints, the information is eventually accurate.

7 Related work

The failure detector presented in this paper is based on seminal work on heartbeat failure detectors by Aguilera, Chen and Toueg [1, 2]. The distributed system model is the same and is suitable for mobile ad hoc networks: An asynchronous model augmented with failure detection for partitionable networks. Even if not mentioned in their papers, heartbeat failure detection algorithms are inherently convenient for tolerating nodes mobility. We have just implicitly added an neighbourhood detector that notifies topology changes of the neighbourhood, and modified the transmission and the parsing of paths in heartbeat messages in order to (1) let the heartbeat message return to the initiator even if the path includes a cycle and to (2) build the set of processes reachable through neighbours.

In the literature, few unreliable failure detectors explicitly target mobile ad hoc networks. In [6], the authors propose an adaptation of a gossip based failure detection. Heartbeat messages are logically stamped, and a vector informing about the last heartbeat messages received, is piggybacked in every heartbeat message. In order to tolerate nodes mobility, the failure detector algorithm allows gaps of some heartbeats between adjacent heartbeat arrivals. Contrary to our distributed model, the links are bidirectional.

In [8], an architecture for local failure detection and tolerating terminal mobility is presented in the context of sensor networks. The distributed model assumes that every process uses the same time unit. The mobility is said to be passive, that is nodes are not aware they are moving, and the network does not partition. In our work, the distributed model is weaker, but assumes that each node is equipped with a neighbour detector. We do not build such neighbour detector but claim that network protocols can provide such information. Furthermore, in [8], the architecture is built around a local failure detector (correct processes eventually only suspect processes in the local neighbourhood) and mobility detection layer. The local failure detection is implemented using any $\diamond\mathcal{P}$ algorithm, not provided in the paper. The mobility detection algorithm is a periodic gossip diffusing computation with one initiator and with termination detection.

Concerning disconnection detection, to the best of our knowledge, there is no deterministic algorithm in the literature. Concerning partition detection, [3] discusses the property of a failure detector for partitionable group communication systems, but the authors do not give any implementation. Our partition detector is inspired by this specification. In addition, contrary to [3], in our proposition, communication channels can be unidirectional.

We see our partition detector as the participant detector introduced in [4]: Take the set of reachable processes as the set of participants and relaunch the consensus in case of the triggering of partition change. In our proposition, the partition detector implicitly relies on a neighbourhood detector. Our neighbourhood detector conforms to the information accuracy property of the participant detector. The second property, namely information inclusion, is not present in our proposition because, contrary to [4], the set of potential participants Π is known.

8 Conclusion and future work

This paper has presented a derived version of a heartbeat failure detector in which the paths of processes piggybacked in the heartbeat messages are also parsed to build a topology of the network reachable through the neighbours. We also used this failure detector to build a quiescent stubborn broadcast primitive. In addition, the failure detector tolerates the mobility of the processes, and then topology changes. Since disconnections are frequent in mobile ad hoc networks, and in order to make the distinction between failures and disconnections, the paper has also introduced the concept of unreliable disconnection detection and has presented a vector-based disconnection detector. Our disconnection detector makes the benefit of the quiescent stubborn channels to diffuse disconnection information whenever possible, that is optimistically. The hints provided by the heartbeat failure detector and by the vector-based disconnection detector are interpreted by an eventually perfect unreliable partition detector for wireless systems subject to node and link crashes failure, and subject to nodes mobility. A first version of the partition detector outputs for each process p , a list of processes suspected to be unreachable, that is those processes which are suspected of being in another partition than p 's one. A second version of the partition detector also constructs, over the correct processes of p 's partition, a convergence on the set of processes suspected to be faulty, the set of processes seen as disconnected, and the set of processes suspected to be partitioned. The goal of characterising the cause of the partition is to distinguish countermeasures for fault tolerance and for disconnection tolerance.

The first perspective to this work is the simulation of the protocols' performance for mobile ad hoc networks and the comparison with the results presented in [6]. Another perspective is the study of the effect of the threshold of our eventually perfect partition detector: In algorithm 3, the threshold is fixed to 1. Last but not least, we plan to use the partition detector for establishing consensus for partitionable networks, that is on a set of participants smaller than Π .

References

- [1] M. Aguilera, W. Chen, and S. Toueg. Using the Heartbeat Failure Detector for Quiescent Reliable Communication and Consensus in Partitionable Networks. *Theoretical Computer Science*, 220(1):3–30, June 1999.

-
- [2] M. Aguilera, W. Chen, and S. Toueg. On Quiescent Reliable Communication. *SIAM Journal of Computing*, 29(6):2040–2073, Apr. 2000.
 - [3] Ö. Babaoğlu, R. Davoli, and A. Montresor. Group Communication in Partitionable Systems: Specification and Algorithms. *IEEE Transactions on Software Engineering*, 27(4):308–336, Apr. 2001.
 - [4] D. Cavin, Y. Sasson, and A. Schiper. Reaching Agreement with Unknown Participants in Mobile Self-Organized Networks in Spite of Process Crashes. Technical Report IC/2005/026, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, 2005.
 - [5] T. D. Chandra and S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225–267, Mar. 1996.
 - [6] R. Friedman and G. Tcharny. Evaluating Failure Detection in Mobile Ad-Hoc Networks. *International Journal of Wireless and Mobile Computing*, 1(8), 2005.
 - [7] R. Guerraoui, R. Oliveira, and A. Schiper. Stubborn Communication Channels. Technical Report TR97, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, 1997.
 - [8] N. Sridhar. Decentralized Local Failure Detection in Dynamic Distributed Systems. In *Proc. 25th IEEE Symposium on Reliable Distributed Systems*, pages 143–154, Leeds (UK), Oct. 2006.
 - [9] L. Temal and D. Conan. Failure, connectivity, and disconnection detectors. In *Proc. 1st French-speaking Conference on Mobility and Ubiquity Computing*, volume 64 of *ACM International Conference Proceeding Series*, pages 90–97, Nice, France, June 2004.



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

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)

Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

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