# An Unreliable Failure Detector for Unknown and Mobile Networks

Pierre Sens[1], Luciana Arantes[1], Mathieu Bouillaguet[1], Véronique Simon[1] , and Fabíola Greve[2]

[1] LIP6, Université Pierre et Marie Curie, INRIA, CNRS, France
{pierre.sens,luciana.arantes,mathieu.bouillaguet,veronique.simon}@lip6.fr
[2] DCC - Computer Science Department / Federal University of Bahia, Brazil
fabiola@dcc.ufba.br

**Abstract.** This paper presents an asynchronous implementation of a failure detector for unknown and mobile networks. Our approach does not rely on timers. Neither the composition nor the number of nodes in the system are known. Our algorithm can implement failure detectors of class $\Diamond S$ when behavioral properties and connectivity conditions are satisfied by the underlying system.

## 1 Introduction

*Unreliable failure detector*, namely FD, is a fundamental service, able to help in the development of fault-tolerant distributed systems. FD can informally be seen as a per process oracle, which periodically provides a list of processes suspected of having crashed. In this paper, we are interested in the class of FD denoted $\Diamond S$ [1]. They ensure that (i) eventually each crashed process is suspected by every correct process (*strong completeness*), and (ii) there is a time after which some correct processes are never suspected (*eventual weak accuracy*).

We propose a new asynchronous FD algorithm for dynamic systems of mobile and unknown networks. It does not rely on timers to detect failures and no knowledge about the system composition nor its cardinality are required. The basic principle of our FD is the flooding of failure suspicion information over the network. Initially, each node only knows itself. Then, it periodically exchanges a QUERY-RESPONSE [2] pair of messages with its neighbors. Based only on the reception of these messages and on the partial knowledge about its neighborhood, a node is able to suspect other processes or revoke a suspicion in the system. A proof that our implementation provides a FD of class $\Diamond S$ is available at the research report [3].

## 2 Model and Behavioral properties

***Model.*** We consider a dynamic distributed system composed of a finite set $\Pi$ of $n > 1$ mobile nodes, $\Pi = \{p_1, \ldots, p_n\}$. Each process knows its own identity and it knows only a subset of processes in $\Pi$. It does not know $n$. There is one process per node which communicates with its 1-hop neighbors by sending and receiving messages via a packet radio network. There are no assumptions on the relative speed of processes or on message transfer delays, thus the system

is asynchronous. A process can fail by crashing. Communications between 1-hop neighbors are considered to be reliable. Nodes are mobile and they can keep continuously moving and pausing. A faulty node will eventually crash. Nonetheless, we assume that there are no network partitions in the system in spite of node failures and mobility. We also assume that each node has at least $d$ neighbors and that $d$ is known to every process. Let $f_i$ denote the maximum number of processes that may crash in the neighborhood of any process. We assume that the local parameter $f_i$ is known to every process $p_i$ and $f_i + 1 < d$.

***Behavioral properties.*** Let us now define some behavioral properties that the system should satisfy in order to ensure that our algorithm implements a FD of class $\Diamond S$. In order to implement any type of FD with an unknown membership, processes should interact with some others to be known. According to [4], if there is some process in the system such that the rest of processes have no knowledge whatsoever of its identity, there is no algorithm that implements a FD with weak completeness. Thus, the following *membership property*, namely $\mathcal{MP}$, should be ensured by all nodes in the system. This property states that, to be part of the membership of the system, a process $p_m$ (either correct or not) should interact at least once with other processes in its neighborhood by broadcasting a QUERY message when it joins the network. Moreover, this query should be received and kept in the state of at least one correct process in the system, beyond the process $p_m$ itself.

Let $p_m$ be a mobile node. Notice that a node can keep continuously moving and pausing, or eventually it crashes. Nonetheless, we consider that, infinitively often, $p_m$ should stay within its target range destination for a sufficient period of time in order to be able to update its state with recent information regarding failure suspicions and mistakes. Hence, in order to capture this notion of "sufficient time of connection within its target range", the following *mobility property*, namely $\mathcal{M}obi\mathcal{P}$, has been defined. This property should be satisfied by all mobile nodes. Thus, $\mathcal{M}obi\mathcal{P}$ for $p_m$ at time $t$ ensures that, after reaching a target destination, there will be a time $t$ at which process $p_m$ should have received QUERY messages from at least one correct process, beyond itself. Since QUERY messages carry the state of suspicions and mistakes in the membership, this property ensures that process $p_m$ will update its state with recent informations.

Let us define another important property in order to implement a $\Diamond S$ FD. It is the *responsiveness property*, namely $\mathcal{RP}$, which denotes the ability of a node to reply to a QUERY among the first nodes. This property should hold for at least one correct node. The $\mathcal{RP}(p_i)$ property states that after a finite time $u$, the set of responses received by any neighbor of $p_i$ to its last QUERY always includes a response from $p_i$. Moreover, as node can move, the $\mathcal{RP}(p_i)$ also states that neighbors of $p_i$ eventually stop moving outside $p_i$'s transmission range. $\mathcal{RP}$ property should hold for at least one correct stationary node. It imposes that eventually there is some "stabilizing" region where the neighborhood of some correct "fast" node $p_i$ does not change.

Properties $\mathcal{MP}$ and $\mathcal{RP}$ may seem strong, but in practice they should just hold during the time the application needs the strong completeness and eventual

weak accuracy properties of FD of class $\diamondsuit S$, as for instance, the time to execute a consensus algorithm.

## 3 Implementation of a Failure Detector of Class $\diamondsuit S$

The following algorithm describes our protocol for implementing a FD of class $\diamondsuit S$ when the underlying system satisfies $\mathcal{MP}$ and $\mathcal{MobiP}$ for all participating nodes and the $\mathcal{RP}$ for at least one correct node. We use the following notations:

- $susp_i$: denotes the current set of processes suspected of being faulty by $p_i$. Each element of this set is a tuple of the form $\langle id, ct\rangle$, where $id$ is the identifier of the suspected node and $ct$ is the tag associated to this information.
- $mist_i$: denotes the set of nodes which were previously suspected of being faulty but such suspicions are currently considered to be a mistake. Similar to the $susp_i$ set, the $mist_i$ is composed of tuples of the form $\langle id, ct\rangle$.
- $rec\_from_i$: denotes the set of nodes from which $p_i$ has received responses to its last QUERY message.
- $known_i$: denotes the current knowledge of $p_i$ about its neighborhood. $known_i$ is then the set of processes from which $p_i$ has received a QUERY message.
- $Add(set, \langle id, ct\rangle)$: is a function that includes $\langle id, ct\rangle$ in $set$. If an $\langle id, -\rangle$ already exists in $set$, it is replaced by $\langle id, ct\rangle$.

The algorithm is composed of two tasks. Task $T1$ is made up of an infinite loop. At each round, a QUERY message is sent to all nodes of $p_i$'s range neighborhood (line 5). Node $p_i$ waits for at least $d - f_i$ responses, which includes $p_i$'s own response (line 6). Then, $p_i$ detects new suspicions (lines 7-12). It starts suspecting each node $p_j$, not previously suspect, which it knows ($p_j \in known_i$), but from which it does not receive a RESPONSE to its last QUERY. If a previous mistake information related to this new suspected node exists in the mistake set $mist_i$, it is removed from it (line 10) and the suspicion information is then included in $susp_i$ with a tag which is greater than the previous mistake tag (line 9). If $p_j$ is not in the $mist$ set (i.e., it is the first time $p_j$ is suspected), $p_i$ suspected information is tagged with 0 (line 12).

Task $T2$ allows a node to handle the reception of a QUERY message. A QUERY message contains the information about suspected nodes and mistakes kept by the sending node. However, based on the tag associated to each piece of information, the receiving node only takes into account the ones that are more recent than those it already knows. The two loops of task $T2$ respectively handle the information received about suspected nodes (lines 18–24) and about mistaken nodes (lines 25–30). Thus, for each node $p_x$ included in the suspected (respectively, mistake) set of the QUERY message, $p_i$ includes the node $p_x$ in its $susp_i$ (respectively, $mist_i$) set only if the following condition is satisfied: $p_i$ received a more recent information about $p_x$ status (failed or mistaken) than the ones it has in its $susp_i$ and $mist_i$ sets. Furthermore, in the first loop of task $T2$, a new mistake is detected if the receiving node $p_i$ is included in the suspected set of the QUERY message (line 20) with a greater tag. At the end of the task (line 31), $p_i$ sends to the querying node a RESPONSE message.

When a node $p_m$ moves to another destination, $p_m$ will start suspecting the nodes of its old destination since they are in its $known_m$ set.

```
1  init:
2   susp_i ← ∅; mist_i ← ∅ ; known_i ← ∅
3  Task T1:
4  Repeat forever
5    | broadcast QUERY(susp_i, mist_i)
6    | wait until RESPONSE received from at least (d − f_i) processes
7    | For all p_j ∈ known_i \ rec_from_i | ⟨p_j, −⟩ ∉ susp_i do
8    |   | If ⟨p_j, ct⟩ ∈ mist_i
9    |   |   | Add(susp_i, ⟨p_j, ct + 1⟩)
10   |   |   | mist_i = mist_i \ {⟨p_j, −⟩}
11   |   | Else
12   |   |   | Add(susp_i, ⟨p_j, 0⟩)
13 End repeat
14
15 Task T2:
16 Upon reception of QUERY (susp_j ,mist_j) from p_j do
17 known_i ← known_i ∪ {p_j}
18 For all ⟨p_x, ct_x⟩ ∈ susp_j do
19 If ⟨p_x, −⟩ ∉ susp_i ∪ mist_i or (⟨p_x, ct⟩ ∈ susp_i ∪ mist_i and ct < ct_x)
20   | If p_x = p_i
21   |   | Add(mist_i, ⟨p_i, ct_x + 1⟩)
22   | Else
23   |   | Add(susp_i, ⟨p_x, ct_x⟩)
24   |   | mist_i = mist_i \ {⟨p_x, −⟩}
25 For all ⟨p_x, ct_x⟩ ∈ mist_j do
26 If ⟨p_x, −⟩ ∉ susp_i ∪ mist_i or (⟨p_x, ct⟩ ∈ susp_i ∪ mist_i and ct < ct_x)
27   | Add(mist_i, ⟨p_x, ct_x⟩)
28   | susp_i = susp_i \ {⟨p_x, −⟩}
29   | If (p_x ≠ p_j)
30   |   | known_i = known_i \ {p_x}
31 send RESPONSE to p_j
```

Lines 29–30 allow the updating of the *known* sets of both the node $p_m$ and of those nodes that belong to the original destination of $p_m$. For each mistake $\langle p_x, ct_x \rangle$ received from a node $p_j$ such that node $p_i$ keeps an old information about $p_x$, $p_i$ verifies whether $p_x$ is the sending node $p_j$. If they are different, $p_x$ should belong to a remote destination. Thus, process $p_x$ is removed from the local set $known_i$.

## References


1. Chandra, T., Toueg, S.: Unreliable failure detectors for reliable distributed systems. JACM **43**(2) (March 1996) 225–267
2. Mostefaoui, A., Mourgaya, E., Raynal, M.: Asynchronous implementation of failure detectors. In: DSN. (June 2003)
3. Sens, P., Arantes, L., Bouillaguet, M., Greve, F.: Asynchronous implementation of failure detectors with partial connectivity and unknown participants. Research Report 6088, INRIA (01 2007)
4. Fernández, A., Jiménez, E., Arévalo, S.: Minimal system conditions to implement unreliable failure detectors. In: PRDC, IEEE Computer Society (2006) 63–72