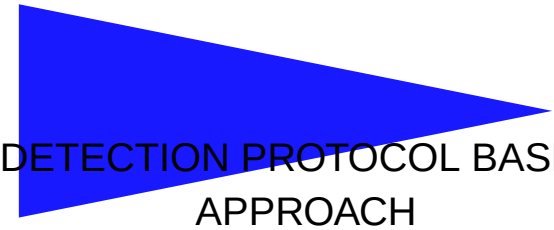
INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

1367

PUBLICATION
INTERNE
N°



A FAILURE DETECTION PROTOCOL BASED ON A LAZY APPROACH

CHRISTOF FETZER, MICHEL RAYNAL, FRÉDÉRIC TRONEL

ISSN 1166-8687

IRISA

CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

A Failure Detection Protocol Based on a Lazy Approach

Christof Fetzer[*] , Michel Raynal[**] , Frédéric Tronel[**]

Thème 1 — Réseaux et systèmes
Projet ADP

Publication interne n ° 1367 — Novembre 2000 — 21 pages

Abstract: The detection of process failures is a crucial problem system designers have to cope with in order to build fault-tolerant distributed platforms. Unfortunately, it is impossible to distinguish with certainty a crashed process from a very slow process in a purely asynchronous distributed system. This prevents some problems to be solved in such systems. That is why failure detector oracles have been introduced to circumvent these impossibility results.

This paper presents a relatively simple protocol that allows a process to "monitor" another process, and consequently to detect its crash. This protocol enjoys the nice property to rely as much as possible on application messages to do this monitoring. Differently from previous process crash detection protocols, it uses control messages only when no application messages is sent by the monitoring process to the observed process. This protocol has noteworthy features. When the underlying system satisfies the partial synchrony assumption, it actually implements an eventually perfect failure detector (i.e., a failure detector of the class usually denoted $\diamond\mathcal{P}$). Moreover, if the upper layer application terminates correctly when the failure detector it uses belongs to $\diamond\mathcal{P}$, then, when run with the proposed protocol, it also terminates correctly. These properties make the protocol attractive: it is inexpensive, implementable, and powerful. The paper also describes performance measurements of an implementation of the protocol.

Keywords:Asynchronous Distributed System, Crash Failure, Eventually Perfect Failure Detector, Experimental Measurement, Failure Detection, Fault-Tolerance, Message-Passing.

(Résumé : tsvp)

[*] christof@att.research.com AT&T, 180 Park Ave, Florham Park, NJ 07932, USA
[**] {raynal, ftronel}@irisa.fr

Résumé : La détection de panne est un problème crucial lors de la conception d'un système réparti qui doit être tolérant au défaillances. Malheureusement il n'est pas possible de résoudre ce problème de manière totalement fiable sans faire d'hypothèses supplémentaires sur les temps d'acheminement des messages au travers du réseau de communication, ainsi que sur la vitesse relative des processus composant le système.

Ce papier présente un protocole très simple qui permet à un processus de surveiller l'activité du système et donc de détecter les éventuelles pannes. Ce protocole utilise autant que possible les messages applicatifs et n'utilise des messages de contrôle ad hoc qu'en dernier recours. Par ailleurs lorsque le système sous-jacent satisfait certaines propriétés de synchronie partielle, ce protocole exhibe un comportement qui le classe dans la catégorie des détecteurs finalement parfait. Ceci implique que si l'application a été conçue avec cette catégorie de détecteurs à l'esprit, elle peut fonctionner en utilisant ce protocole.

Toutes ces propriétés rendent ce protocole particulièrement efficace et peu coûteux. Des mesures de performances dans diverses topologie de réseaux sont données.

Mots-clés: Système réparti asynchrone, Défaillances, Détecteur de défaillances finalement parfaits.

1 Introduction

Context of the study and related work The design of fault-tolerant middleware on top of asynchronous distributed systems prone to process crash failures is an important problem system designers have to cope with. Unfortunately, due to the effect of asynchrony and process crashes, some problems are impossible to solve in purely asynchronous systems. One of the most famous is the *Consensus* problem which cannot be solved by a deterministic protocol if (even a single) process can crash [6].

This makes the detection of process crash failures a central problem of fault-tolerant distributed computing. As a crashed process cannot be distinguished with certainty from a slow process or from a process with which communication are very slow, some authors have introduced the idea of *Unreliable Failure Detector* oracle [2]. Such an oracle provides each process with the list of processes it suspects to have crashed. According to the properties their guesses have to satisfy, several classes of failure detectors have been defined. One of them includes all the failure detectors that, after some unknown but finite time, make no mistake. Hence, these failure detectors guarantee that, after some time, the lists of suspects include all crashed processes and no non-crashed process. This class, denoted $\diamond\mathcal{P}$, is called the class of *Eventually Perfect Failure Detectors*. (Failure detectors with different aims and properties are studied in [1, 7].)

Unfortunately, except for some very particular failure detectors [8], the major part of "reasonable" failure detectors cannot be implemented in asynchronous distributed systems. If they were, their implementation would contradict impossibility results (e.g. [6]). Hence, the idea to enrich the underlying system with additional assumptions in order that a failure detector of the class required to solve the problem we are concerned with (e.g., [2, 9]) becomes implementable. This means that the problem can be solved when the system satisfies these assumptions.

Let us consider systems that, even if they behave in a totally asynchronous way during some time, eventually behave in a synchronous way. More precisely, this means that there are bounds on message transfer delays, but these bounds are not known and they hold only after some unknown time [2, 5]. Considering such *partially synchronous* systems, some authors have provided implementations of $\diamond\mathcal{P}$ [2, 10] or $\diamond\mathcal{S}$ [11] ($\diamond\mathcal{S}$ is the weakest failure detector class that allows to solve the consensus problem [3]. Several $\diamond\mathcal{S}$-based consensus protocols are designed, e.g., [2, 13].)

The protocol implementing a failure detector of the class $\diamond\mathcal{P}$ described in [2], and the protocol implementing a failure detector of the class $\diamond\mathcal{S}$ described in [11] are based on a *gossiping* approach. Repeatedly, a process (or several processes) sends "I am alive" messages to a subset or the whole set of the processes. Differently, the modular suite of protocols implementing several classes of failure detectors (among which $\diamond\mathcal{P}$ and $\diamond\mathcal{S}$) introduced in [10] is based on a *monitoring* approach. Periodically, a process sends an "Are you here" message to another process (the process it monitors) which has to send back an "I am alive" message. Moreover, the processes are placed on a logical ring which defines the monitoring and the failure information propagation pattern. When compared to the gossiping approach, this monitoring approach reduces the cost of failure detection (expressed in number of control messages).

Paper content This paper is on the detection of process failures. The previous (gossiping or monitoring-based) failure detectors protocols have a major drawback: they have been designed by taking into account properties the underlying system is assumed to satisfy, but without considering the behavior of the upper layer protocol that uses the failure detector they implement. To be more explicit, in any partially synchronous distributed system, they build a failure detector with the required properties by permanently sending and processing control messages. But, proceeding that

way, they send control messages and consume resources even when the failure detector is not used by the upper layer protocol!

This paper proposes a new and simple approach to implement a failure detector. The novelty comes from the fact that the cost associated with the implementation of a failure detector is paid only when the failure detector is used (hence the name *Lazy* approach). More precisely, this paper proposes a protocol that allows a process to monitor another process. If these processes are communicating, the application messages they exchange are used to save failure detection messages. Only when they are not communicating, failure detection messages are used. Hence, the protocol is intimately related to communication. This basic protocol ensures that if a process queries another process that has crashed, then it will definitely suspect it (completeness of the detection).

The proposed failure detection protocol (denoted $\mathcal{FD}_L$) is then plugged into two particular contexts. The first context is defined by properties assumed to be satisfied by the "lower layer" (i.e., the underlying system); the second context is defined by properties assumed to be satisfied by the "upper layer" (i.e., the application) [1]. More specifically, the first context is defined by a behavioral property the underlying system has to satisfy, namely, partial synchrony. It is shown that, when plugged in such a system, the proposed protocol provides a failure detector of the class $\diamond\mathcal{P}$. The second context that we consider is defined by a property of the application that uses the protocol. Let a failure detector-based application be $\diamond\mathcal{P}$-*terminating* if it terminates correctly (within at most some l steps after the failure detector becomes perfect) when the failure detector it uses belongs to $\diamond\mathcal{P}$. It is shown that, when run with a $\diamond\mathcal{P}$-terminating application, the protocol actually provides the application with the same properties as $\diamond\mathcal{P}$. Interestingly and differently from the first context, this second context does not require an a posteriori upper bound on message transfer delays. These two contexts shows both the theoretical interest and the wide practical applicability of the proposed failure detection protocol. It enjoys the following nice features: it is inexpensive, implementable and powerful.

Paper organization The paper is made up of seven sections. Section 2 defines the basic distributed system model. Section 3 presents the failure detection protocol and proves that it detects crashed processes (completeness property of the detection). Then, Sections 4 and 5 present the two contexts previously suggested and show the additional properties offered by the protocol in each of them. Section 6 describes performance measurements obtained by a protocol implementation. Finally, Section 7 concludes the paper.

2 Underlying Basic Distributed System

Processes with crash failure The basic system consists of a finite set Π of $n > 1$ processes, namely, $\Pi = \{p_1, p_2, \ldots, p_n\}$. A process can fail by *crashing*, *i.e.*, by prematurely halting. It behaves correctly (*i.e.*, according to its specification) until it (possibly) crashes. By definition, a *correct* process is a process that does not crash. Each process p_i has a local hardware clock hc_i that strictly monotonically increases. The local clocks are not required to be synchronized, and there is no assumption on their possible drift.

The behavior of a process can be modeled by a finite state automaton. We consider that each step of a process is triggered by a message (an internal statement can be modeled as a message sent by a process to itself). Each execution of a communication statement by a process p_i defines an event. The history h_i of a process p_i is the sequence of communication events it produces.

Let $hc(e)$ be the value of hc_i when p_i produces the event e. Let us consider two distinct events $e1$ and $e2$ produced by p_i. It is assumed that the granularity $g > 0$ of hc_i is such $|hc(e2) - hc(e1)| \geq g$.

Communication Processes communicate and synchronize by sending and receiving messages through channels. Every pair of processes is connected by a channel. Channels are not required to be FIFO. They are only assumed to be reliable in the following sense: they do not create, duplicate, alter or lose messages. This means that a message sent by a process p_i to a process p_j is assumed to be eventually received by p_j, if p_j is correct.

The primitives "*send* m *to* p_j" and "*receive* m *from* p_j" are used by a process p_i to communicate with the process p_j.

3 A Lazy Failure Detection Protocol $\mathcal{FD}_L$

3.1 Service Provided to the Upper Layer

The protocol provides the following primitives to each upper layer application process p_i (see Figure 1 where M denotes an application message, m a protocol message).

- SEND M to p_j: used by p_i to send an application message M to p_j.
- RECEIVE M: used by p_i to receive an application message M.
- QUERY(j): used to know whether p_j is suspected to have crashed. This primitive returns an answer, namely, the value *suspect* or *no_suspect*.

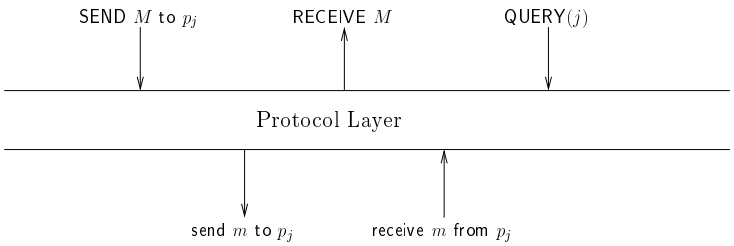


Figure 1: Layered Structure

3.2 Underlying Principles

The principle that underlies the protocol is simple. As indicated in the Introduction, it consists in using as much as possible the application messages to get information on process failures. This requires that each message be acknowledged. Hence, if a process p_i inquires about p_j (by invoking QUERY(j)), the answer depends on the round trip delays of the messages already acknowledged. Differently, if a process p_i inquiries about p_j while it is not communicating with p_j, control messages are used to compensate the absence of application messages.

At a more operational level, the protocol uses three types of messages: "appl", "ack" and "ping". To send an application message M to p_j, a process p_i invokes "*send* appl(m) *to* p_j" where the protocol message m includes M plus some control information. When, it receives such a message, p_j systematically acknowledges it by sending back ack(m). When it receives ack(m), p_i computes the round trip delay of the pair appl(m)+ack(m). In addition, for each destination process p_j, p_i computes the maximum round trip delay for the messages that have been acknowledged by p_j.

As suggested previously, the answer provided by QUERY(j) when it is invoked by the upper layer depends on the existence of "pending" messages, i.e., messages m such that appl(m) has been sent to p_j but the corresponding ack(m) has not yet been received by p_i:
- If there is no such message, the answer is *no_suspect*, but p_i sends a ping message to p_j in order to verify its answer.
- If there are such "pending" messages, the answer depends on the maximum round trip delay already experienced.

3.3 Protocol

For each process p_i, the protocol manages two arrays of local variables:

- $pending\_msg\_st_i[j]$: this set contains the sending times (determined by the local clock hc_i) of the messages sent by p_i to p_j and whose acknowledgments has not yet been received by p_i. This set is initially empty.

  When applied to the set $pending\_msg\_st_i[j]$, the function *min* delivers its smallest value. ($min(\emptyset)$ is undefined.)

- $max\_rdt_i[j]$: contains the biggest round trip time of the messages that p_i has sent to p_j and that have been acknowledged. Initially, this variable has the value 0. In practice, if one knows the value of $max\_rdt_i[j]$ from a previous execution, one can initialize $max\_rdt_i[j]$ to this previous value instead.

The protocol behavior for p_i is described in Figure 2. A call to SEND M is interpreted as a message reception from the upper layer. Similarly, RECEIVE M is interpreted as a message sending to the upper layer. As we have seen, a protocol message m has a type (appl/ack/ping). Moreover, in addition to a content ($m.content$), it also carries a local sending time ($m.st$). More precisely, apply(m) and ping(m) carry their local sending time. Differently, ack(m) carries the sending time of the apply(m) or ping(m) message it is associated with.

3.4 Completeness Property

If from some time t, a process p_i obtains the answer *suspect* each time it invokes QUERY(j), we say that from that time it "permanently suspects p_j" from t.

Theorem 1 *Let us assume that p_i is correct, while p_j is faulty (i.e., it crashes). Then, $\mathcal{FD}_L$ ensures that eventually p_i permanently suspects p_j to have crashed.*

Proof Let us note that after p_j has crashed, it does not send "appl, ack" or "ping" messages. Moreover, the messages it sent to p_i before crashing are eventually received by p_i; let t be an instant after these messages have been received. This means that after t:
- (A0) Line 9 is never executed: consequently $max\_rdt_i[j]$ remains equal to some value R;
- (A1) Line 10 is never executed: consequently no value is suppressed from $pending\_msg\_st_i[j]$.
We show that, after some time $t' \geq t$, all the invocations of QUERY(j) entail the execution of the lines 14, 19-21, and consequently, they all output *suspect*. We consider two cases.

Case 1: At t, there is at least one appl(m) or ping(m) message sent by p_i to p_j that has not been acknowledged by p_j.

```
(1) when SEND M to p_j is invoked:
(2)   m.content ← M; m.st ← hc_i;
(3)   pending_msg_st_i[j] ← pending_msg_st_i[j] ∪ {m.st}
(4)   send appl(m) to p_i

(5) when type(m) is received from p_j:
(6)   case type=appl  then transmit M = m.content to the upper layer; % RECEIVE M %
(7)                        send ack(m) to p_j % m.st keeps its value %
(8)        type=ack   then rt ← hc_i;
(9)                        max_rtd_i[j] ← max(max_rtd_i[j], rt − m.st);
(10)                       pending_msg_st_i[j] ← pending_msg_st_i[j] − {m.st}
(11)       type=ping  then send ack(m) to p_j % m.st keeps its value %
(12)  endcase

(13) when QUERY(j) is invoked:
(14)  if pending_msg_st_i[j] = ∅ then create a control message m;
(15)                               m.content ← null; m.st ← hc_i;
(16)                               send ping(m) to p_j;
(17)                               pending_msg_st_i[j] ← {m.st};
(18)                               return (no_suspect)
(19)                          else rt ← hc_i;
(20)                               if ( (rt−min(pending_msg_st_i[j])) > max_rtd_i[j] )
(21)                                        then return (suspect)
(22)                                        else return (no_suspect)
(23)                               endif
(24)  endif
```

Figure 2: Lazy Failure Detection Protocol

In that case we have after t:
- (A2): Due to (A1) the value $m.st$ remains permanently in the set $pending\_msg\_st_i[j]$ (this value has been added to this set at line 3 or 17, according to the type "appl/ping" of m).
- (A3): Due to (A1) and (A2) $min(pending\_msg\_st_i[j])$ remains forever $\leq m.st$.
Let us consider the execution of an infinite sequence of QUERY(j) invoked by p_i after t. We have:
- (A4): Due to (A2) and the test of line 14, it follows that all those invocations entail the execution of line 19. Let $rt(1), rt(2), \ldots$ be the sequence of values obtained at that line. Due to the monotonicity property of the local clocks, this sequence is monotonically increasing.
- (A5): We conclude from (A4) that there is an integer x such that the following condition $(\forall y \geq x : (rt(y) > m.st + R))$ is true.
We conclude from (A0), (A3) and (A5) that there is a time instant $t' \geq t$ after which the test of line 20 is always satisfied when p_i invokes QUERY(j). Consequently, from t', any invocation of QUERY(j) returns *suspect*.

Case 2: At t, all the appl(m) and ping(m) messages sent by p_i to p_j have been acknowledged. This means that at t we have $pending\_msg\_st_i[j] = \emptyset$. So, let us consider the first QUERY(j) invocation issued by p_i after t. As $pending\_msg\_st_i[j] = \emptyset$, p_i executes the lines 14-18: it creates a control message m, adds $m.st$ to $pending\_msg\_st_i[j]$, and we are now in Case 1. $\Box_{Theorem\ 1}$

3.5 Message Cost of the Protocol

Each appl() or ping() message generates at most one ack() message. Let us observe that both appl() and ping() are due to the application layer, appl() when it sends an application message, ping() when it invokes QUERY().

Let us consider the cost of an invocation of QUERY(j) by a process p_i after p_j has crashed. According to the current state of $pending\_msg\_st_i[j]$, p_i can be forced to send a ping(m) message to p_j. But from now, the condition $pending\_msg\_st_i[j] \neq \emptyset$ remains permanently true. Consequently, the next invocations of QUERY(j) do not send messages, and their communication cost is 0.

4 Plugging $\mathcal{FD}_L$ in a Partially Synchronous System

This section studies the properties offered by the protocol when it is used in partially synchronous systems. As the class $\diamond\mathcal{P}$ (Eventually Perfect failure detectors) is central to these properties, it is first introduced. Moreover, this section assumes that the drift ρ_i of each local clock hc_i is bounded. To simplify the presentation and without loss of generality, we consider the clocks have no drift (see the techniques presented in [4, 12, 14] to take drifts into account).

4.1 The Class of Eventually Perfect Failure Detectors

The class $\diamond\mathcal{P}$ of failure detectors has been introduced in [2]. It is defined by the following properties. Completeness is on the actual detection of failures, Eventual Accuracy limits the mistakes a failure detector can make:

- Completeness: Eventually every correct process permanently suspects every crashed process.
- Eventual Accuracy: Eventually no correct process suspects a correct process.

Let us first observe that Theorem 1 proves that, when used in an asynchronous distributed system (as defined in Section 2), the protocol described in Figure 2 implements the Completeness property. In the following we show that it also implements the Eventual Accuracy property when some additional assumptions are satisfied.

4.2 The Protocol in Partially Synchronous Systems

Let us assume that the underlying system is partially synchronous [2, 5]. As indicated in the Introduction, this means that there is a time (a priori unknown) after which there are upper bounds (not known) on messages transfer delays and associated processing times.

The combination of Theorem 1 and Theorem 2 (that follows) shows that, when used in such a context, the protocol $\mathcal{FD}_L$ actually implements a failure detector that belongs to $\diamond\mathcal{P}$.

Theorem 2 *If the underlying system is partially synchronous, there is a time t after which $\mathcal{FD}_L$ ensures that no correct process is suspected by a correct process.*

Proof In the following p_i and p_j are two correct processes. Let t_{ub} be the time after which there are upper bounds on message transfer delays. Moreover, let t be a time such that $t \geq t_{ub}$ and after which the ack(m) messages sent by p_j to p_i associated with the appl(m) and ping(m) messages sent by p_i to p_j before t_{ub}, have been received.

We claim (claim 1) that, $\exists t' \geq t$ such that the condition $(rt - min(pending\_msg\_st_i[j]) > max\_rtd_i[j])$ is never satisfied after t'. It follows from the claim that line 21 is never executed after t', which proves the theorem.

Proof of claim 1. From the definition of t, we can conclude that after t the round trip delays between p_i and p_j have an upper bound $\Delta_{i,j}$.

Let us consider the value of $min(pending\_msg\_st_i[j])$ when line 20 is executed at or after t. Let us first note that, as this line is in the **else** part, we have $pending\_msg\_st_i[j] \neq \emptyset$. Let m be such that $m.st = min(pending\_msg\_st_i[j])$. It corresponds to an appl(m) or ping(m) message sent by p_i to p_j after t (at line 4 or 16) and not yet acknowledged. Due to (1): the bound $\Delta_{i,j}$, (2): the fact that ack(m) is not yet received but will be received, and (3): the fact that rt is the current time value, it follows that $(rt - m.ts < RT_m - m.st \leq \Delta_{i,j})$ (where RT_m is p_i's local time at which ack(m) will be received). We now consider two cases.

- Case 1: at t, $max\_rtd_i[j] \geq \Delta_{i,j}$. In that case, we have $(rt - m.st < RT_m - m.st \leq \Delta_{i,j} \leq max\_rtd_i[j])$, and the claim follows.

- Case 2: at t, $max\_rtd_i[j] < \Delta_{i,j}$. We claim (claim 2) that, after some time $t' \geq t$, $max\_rtd_i[j]$ remains constant, equal to a value $R \leq \Delta_{i,j}$. This means that R is an upper bound for the round trip delays between p_i and p_j. Hence, we have $RT_m - m.st \leq R$, and the theorem follows.
  Proof of claim 2. Let us note that $max\_rtd_i[j]$ can only increase. By contradiction, let us assume that $max\_rtd_i[j]$ never stops increasing. Due to the granularity g of hc_i, each time it is increased, $max\_rtd_i[j]$ is increased by at least g (at line 9). The sequence of values taken by $(\Delta_{i,j} - max\_rtd_i[j])$ is monotonically decreasing and eventually becomes negative. A contradiction, as $\Delta_{i,j}$ is an upper bound for the round trip delays.

$\square_{Theorem\ 2}$

5 The Case of $\diamond\mathcal{P}$-Terminating Protocols

5.1 Lazy Failure Detectors

The behavior of a failure detector belonging to a class defined in [2] is actually "eager" in the sense that a failure detector can change its output value even if it is not queried. As we have seen, the present paper introduced the notion of "lazy" failure detector. Here "lazy" is opposed to "eager" and means that the output of the failure detector (1) can not be observed without querying it and (2) is restricted to the process that is queried.

The previous section has shown that when the system is partially synchronous, the "lazy" failure detector $\mathcal{FD}_L$ actually allows to build a failure detector of $\diamond\mathcal{P}$. The aim of this section is to show that the same result can be obtained when the application and the underlying system satisfy some reasonable assumptions. More precisely, we consider the class of protocols that terminate when there is no bound on the length of periods without erroneous suspicions, that is, eventually the protocol will find a sufficiently long period to terminate during which the failure detector behaves without making mistakes. Interestingly, this property is weaker than the properties defining partial synchrony (as defined in the previous section), e.g., they do not require upper bounds on message transfer delays.

We call this class of protocols $\diamond\mathcal{P}$-*Terminating* protocols. We show that these protocols terminate with probability 1 when using $\mathcal{FD}_L$ instead of a protocol belonging to $\diamond\mathcal{P}$. The proof is by contradiction, namely it is shown that infinite executions that exhibit a bound on the length of periods without erroneous suspicions, occur with probability 0 (under the assumptions made on the systems).

The content of this long section (6 pages) is basically the proof of Theorem 3. To be precise and prevent ambiguities, this section uses some formalism, and consequently, is more formal than the other sections. (This is the "price" that has to be paid to get a proof !).

5.2 Failure Patterns and Failure Detector Histories

The notions introduced in this section are from [2, 3]. A failure pattern is a function $F : \mathbb{R} \to \mathcal{P}(\Pi)$, where $F(t)$ is the set of crashed processes at time t. Since there is no process recovery F is not decreasing, that is $\forall t < t' \in \mathbb{R} \quad F(t) \subseteq F(t')$. We define two predicates, *correct*(p) and *faulty*(p) as follows: $faulty(p) \Leftrightarrow \exists t \in \mathbb{R}$ s.t. $\forall t' > t \quad p \in F(t')$ and *correct*$(p) = \neg$*faulty*(p). Let $\mathcal{F}$ be the set of failure patterns.

A failure detector history H with range $\mathcal{V}$ is a function $H : \Pi \times \mathbb{R} \to \mathcal{V}$. Let $\mathcal{H}_\mathcal{V}$ be the set of failure detector histories with range $\mathcal{V}$.

A failure detector D with range $\mathcal{V}$ is a function that maps a failure pattern to a set of failure detector histories with range $\mathcal{V}$. That is $D : \mathcal{F} \to \mathcal{P}(\mathcal{H}_\mathcal{V})$.

The class $\diamond P$ of eventually perfect failure detectors [2] has been informally defined in Section 4.1. More formally, it is the set of failure detectors D with range equal to $\mathcal{P}(\Pi)$ that satisfy the following properties:

$$\begin{cases} (A) & \forall F \in \mathcal{F} \quad \forall H \in D(F) \quad \forall p \in \Pi \quad \textit{faulty(p)} \Rightarrow \exists\, t_0 \in \mathbb{R} \quad \forall t > t_0 \in \mathbb{R} \\ & (\forall q \in \Pi \text{ s.t. } \textit{correct}(q)) \quad p \in H(q,t), \\ (C) & \forall F \in \mathcal{F} \quad \forall H \in D(F) \quad \forall p \in \Pi \quad \textit{correct(p)} \Rightarrow \exists\, t_1 \in \mathbb{R} \quad \forall t > t_1 \in \mathbb{R} \\ & (\forall q \in \Pi \text{ s.t. } \textit{correct}(q)) \quad p \notin H(q,t). \end{cases}$$

As we want to prove that the protocols designed with $\diamond\mathcal{P}$ in mind also terminate when using the $\mathcal{FD}_L$ failure detector, we have to simulate the interface provided by $\diamond\mathcal{P}$. Two situations can be considered. A process p_i queries its $\diamond\mathcal{P}$ module either to check if a particular process namely p_j is suspected, or to know the whole set of processes that are currently suspected. In the first case, the simulation is done by a simple call QUERY(j). In contrast, the second case requires to query the lazy failure detector for each process in the system. Whatever the case, we keep on describing the system model for protocols using a failure detector belonging to the $\diamond\mathcal{P}$ class, as this service can be simulated by $\mathcal{FD}_L$.

5.3 Protocol Automaton and Protocol Histories

Each process is a state automaton (finite or infinite) whose transitions are triggered by the arrival of a message[1]. Moreover, let $\mathcal{M}$ be the set of messages that can be exchanged. To each message $m \in \mathcal{M}$ is associated the following control information:
- $m.st$ is the sending time of m (already defined in 3.3),
- $m.rt$ is the receive time of m,
- $m.src$ is the identity of the process that has sent m,
- $m.dst$ is the identity of the process to which m has been sent.

A protocol automaton using a failure detector, whose output is a set of values $\mathcal{V}$, is a 4-tuple (S, S_0, S_f, δ) where S is the set of all the states that the automaton can enter, $S_0 \subset S$ is the set of initial states, $S_f \subset S$ is the set of final states.

Let $k \geq n$ be a positive integer and $\mathcal{P}_k(\mathcal{M})$ be the set of all subsets that contain at most k messages. The transition function δ is formally defined by:

$$\begin{cases} \delta : S \times \mathcal{V} \times \mathcal{M} \rightarrow S \times \mathcal{P}_k(\mathcal{M}), \\ \delta(s_1, v, m) = (s_2, \{m_1, \ldots, m_l\}). \end{cases}$$

It means that on the receipt of message m, while in state s_1, obtaining v as value of failure detection, a process p enters state s_2 and sends the (possibly empty) set of messages $\{m_1, m_2, \ldots, m_l\}$ $(l \leq k)$.

An event e is defined by a 5-tuple $(p_i, t, s, v, m) \in \Pi \times \mathbb{R} \times S \times \mathcal{V} \times \mathcal{M}$, where p_i is the identity of the process at which the event occurs, t is the real time at which the event occurs, v is the value returned by the failure detector at time t, s is the state in which p_i was before e and m is the message that has triggered m. In particular this means that $m.rt = t$. One can deduce that event e will trigger the transition where p_i is going to enter state s' and send the set of messages $\{m_1, \ldots, m_l\}$ defined by $(s', \{m_1, \ldots, m_l\}) = \delta(s, v, m)$. Let us note that $m_1.st = \ldots = m_k.st = t$. That means that we consider computation time to be part of the transmission time of messages.

A valid protocol history h is a sequence of events $(e_i)_{i \in I}$ such that: (H1) all the events in the sequence are totally ordered by their time of occurrence; (H2) Either a process starts from a given initial state and takes its first step in the protocol during an event triggered by a special fake message called *start*, or is initially crashed and thus takes no steps in the sequence; (H3) Each event of the history is triggered by a single message that has been sent by a given process during a previous event of the history, or is the first event taken by a process; (H4) A correct process takes an infinite number of steps or it terminates; (H5) A process that terminates stays terminated[2];

[1] Transitions can not be triggered by passage of time, since we consider purely time-free applications.

[2] We allow a terminated process to receive messages and to send messages to permit the failure detector to acknowledge ping and application messages.

(H6) Each message sent by a correct process to another correct process is eventually received. This can be formally expressed as follows:

$$
\begin{array}{ll}
(H1) & \forall i \in I \quad \forall j \in I \quad j < i \wedge e_i = (\_, t_1, \_, \_, \_) \wedge e_j = (\_, t_2, \_, \_, \_) \Rightarrow t_1 > t_2, \\
(H2) & \forall p \in \Pi \quad (\exists\, i \ \in I \quad e_i = (p, \_, s, \_, start) \wedge s \in S_0 \wedge (\nexists\, j \in I \quad j < i \wedge e_j = (p, \_, \_, \_, \_))) \\
 & \vee (\nexists\, i \in I \quad e_i = (p, \_, \_, \_, \_)), \\
(H3) & \forall i \in I \quad e_i = (p, \_, \_, \_, m_1) \Rightarrow (\exists\, j \in I \quad j < i \wedge (e_j = (q, \_, s_2, v_2, m_2)) \\
 & \wedge (\delta(s_2, v_2, m_2) = (\_, M)) \wedge (m_1 \in M) \wedge (m_1.dst = p) \wedge (m_1.src = q)) \\
 & \vee (m_1 = start \wedge \nexists\, j < i \quad e_j = (p, \_, \_, \_, \_)), \\
(H4) & \forall p \in \Pi \quad correct(p) \Rightarrow \forall i \in I \quad \exists\, j \in I \quad (j > i \wedge e_j = (p, \_, \_, \_, \_)) \\
 & \vee (\exists s_f \in S_f \wedge e_j = (p, \_, s_f, \_, \_)), \\
(H5) & \forall i \in I \quad \forall p \in \Pi \quad \forall s_f \in S_f \quad correct(p) \wedge e_i = (p, \_, s_f, v, m) \\
 & \Rightarrow \exists s_{f1} \in S_f \quad \exists M : \delta(s_f, v, m) = (s_{f1}, M), \\
(H6) & \forall p, q \in \Pi \quad (correct(p) \wedge correct(q)) \quad \forall i \in I \quad (e_i = (p, \_, s, v, m)) \wedge (\delta(s, v, m) = (\_, M)) \\
 & \wedge (m' \in M) \wedge (m'.dst = q) \Rightarrow (\exists\, j \in I \quad j > i \quad e_j = (q, \_, \_, \_, m')).
\end{array}
$$

We say that a failure detector is *perfect* during a subsequence of events S iff during S the failure detector suspects all processes that have crashed and it does not suspect any processes that have not crashed.

Definition 1 (PERFECT$_l^j$ FAILURE DETECTION PREDICATE) *Let F be a failure pattern and $h = (e_i)_{i \in I}$ a valid protocol history for F. The subsequence S of l consecutive events of h starting at index j $(e_i)_{j \leq i < j+l}$ satisfies the* PERFECT$_l^j$ *predicate if and only if:*

$$
\forall (p, t, s, v, m) \in S \begin{cases} (PC) & \forall q \in \Pi \quad s.t.\ q \in F(t) \Rightarrow q \in v, \\ (PA) & \forall q \in \Pi \quad s.t.\ q \notin F(t) \Rightarrow q \notin v. \end{cases}
$$

5.4 System and Application Requirements

A protocol is terminated by event e_k if all processes have either terminated or crashed no later than event e_k.

Definition 2 (TERMINATED^k TERMINATION PREDICATE) *Let F be a failure pattern, let $h = (e_i)_{i \in I}$ be a valid protocol history for F. The history satisfies predicate* TERMINATED^k*, i.e., has terminated by event $e_k = (\_, t, \_, \_, \_), k \in I$ if and only if $\forall p \in \Pi : (\exists i \leq k \wedge \exists s_f \in S_f : e_i = (p, \_, s_f, \_, \_)) \vee p \in F(t)$.*

We say that a protocol is $\diamond\mathcal{P}$-TERMINATING if there exists a positive integer l such that the protocol terminates whenever the failure detector is perfect for at least l steps.

Definition 3 ($\diamond\mathcal{P}$-TERMINATING PROTOCOLS) *Let $\mathcal{AP}$ be a protocol using a failure detector with range $\mathcal{P}(\Pi)$. $\mathcal{AP}$ is said to be $\diamond\mathcal{P}$-terminating if and only if for each valid protocol history h of $\mathcal{AP}$:*

$$
\exists\, l_0 \in \mathbb{N} \quad \forall l > l_0 \qquad \forall\, j \in \mathbb{N} \quad \text{PERFECT}_l^j \Rightarrow \text{TERMINATED}^{j+l}.
$$

In the rest of the paper we consider particular sequences of messages that are said to be *overlapping*. Informally, this means that we only consider totally ordered (by receipt time) sequences of messages where, each time a message is received, there is at least one other message (of the sequence) that is in transit. More formally:

Definition 4 (RTD(m) ROUND-TRIP DELAY) *Let m be an application message and be $ack(m)$ the acknowledgment of m. The round-trip delay of m is defined as: $RTD(m) = ack(m).rt - m.st$. We define $RTD(ack(m)) = 0$.*

Definition 5 (SEQUENCE OF OVERLAPPING MESSAGES) *Let $M = (m_i)_{i \in I}$ be a sequence of messages, where I is a totally ordered countable (finite or infinite) set of indexes. M is said to be a* sequence of overlapping messages *if :*

$$\forall i \in I \begin{cases} (O1) & RTD(m_i) < \infty, \\ (O2) & \forall\, j \in I \quad j > i \qquad m_i.rt < m_j.rt, \\ (O3) & \exists\, j \in I \quad j > i \qquad m_j.st \leq m_i.rt \text{ unless } i = \max(I). \end{cases}$$

Let us remark that in the case of a finite sequence, conditions (O2) and (O3) do not constraint the last message of the sequence. Furthermore conditions (O2) and (O3) imply that $\forall\, i \in I \quad \exists\, j \in I \quad j > i \quad m_j.st \leq m_i.rt < m_j.rt$ unless $i = \max(I)$. This means that there is at least one message (namely m_j) that is in transit by the time m_i is received. (Figure 3 depicts such a sequence of length 4.)

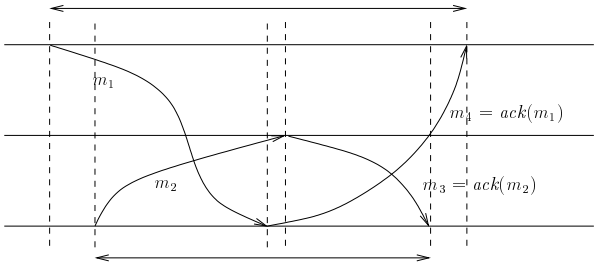


Figure 3: Overlapping Messages

Definition 6 (SEQUENCE OF OVERLAPPING EVENTS) *Let h be a protocol history. Let $S = (e_i)_{i \in I}$ be a subsequence of events extracted from h. S is said to be a* sequence of overlapping events *if the sequence $M = (m_i)_{i \in I}$ of triggering messages associated with S and defined by $m_i \in M \Leftrightarrow e_i = (p, t, s, v, m_i) \in S$ is overlapping.*

Assumption 1 (AVERAGE ROUND TRIP DELAY OF OVERLAPPING SEQUENCES) *Let $S = (m_i)_{i \in I}$ be an overlapping sequence of messages. The average round trip delay of S is defined as $ARTD(S) = \frac{\sum_{i \in I} RTD(m_i)}{|I|}$. We assume that every overlapping sequence of messages has a finite mean and variance round trip delay.*

5.5 Property of $\mathcal{FD}_L$

Theorem 3 *If the upper layer protocol is $\diamond\mathcal{P}$-Terminating, then it terminates correctly when, instead of using a failure detector of $\diamond\mathcal{P}$, it uses $\mathcal{FD}_L$.*

Lemma 1 *Let $A = (a_1, \ldots, a_u)$ and $B = (b_1, \ldots, b_v)$ be two overlapping sequences of messages such that $a_u = b_1$. Let $C = (a_1, \ldots, a_u = b_1, \ldots, b_v)$ be the concatenation of A and B. Then C is overlapping.*

Proof We use the following notation $C = (c_i)_{1 \leq i \leq u+v-1}$, where $c_i = a_i$ if $i \leq u$ and $c_i = b_{i-u+1}$ if $u \leq i \leq u+v-1$. It is obvious that for each $i \in [1, u+v-1]$, one has $RTD(c_i) < \infty$, since this was true for all a_i and b_i. The same argument applies for condition $(O2)$. Finally, condition $(O3)$ is satisfied since:

$$\begin{cases} \text{if } i < u, & \text{one has } c_i = a_i \text{ and } \exists j \leq u \quad a_j.st \leq a_i.rt, \\ \text{if } u \leq i < u+v-1, & \text{one has } c_i = b_{i-u+1} \text{ and } \exists j \leq v \quad b_j.st \leq b_i.rt \\ & \text{that is } \exists j' \quad u < j' \leq u+v-1 \quad (c_{j'} = b_j).st \leq (c_i = b_{i-u+1}).rt, \\ \text{if } i = u+v-1, & \text{one has } i = max(I) \text{ and hence no property to satisfy.} \end{cases}$$

$\square_{Lemma\ 1}$

Lemma 2 *Let $A = (a_1, \ldots, a_u)$ and $B = (b_1, b_v)$ be two overlapping sequences of messages such that $a_u = b_v$. Let $C = sort(A, B)$ be the sequence of messages built from A and B by sorting messages according to their receive time. Then C is overlapping.*

Proof We use the same notation as in Lemma 1. It is obvious that for each $i \in [1, u+v-1]$, one has $RTD(c_i) < \infty$, since this was true for all a_i and b_i. Condition $(O2)$ is naturally satisfied since the sequence C is sorted by receipt time. Finally, condition $(O3)$ is satisfied since:

$$\begin{cases} \text{if } c_i = a_k \text{ and } i < u+v-1 & \text{one has } \exists j > k \quad a_j.st \leq a_k.rt \\ & \text{that is } \exists j' > i \quad (c_{j'} = a_j).st \leq (a_k = c_i).rt, \\ \text{if } c_i = b_k \text{ and } i < u+v-1 & \text{one has } \exists j > k \quad b_j.st \leq b_k.rt \\ & \text{that is } \exists j' > i \quad (c_{j'} = b_j).st \leq (b_k = c_i).rt, \\ \text{if } i = u+v-1 & \text{since } c_{u+v-1} = a_u = b_v, \text{ and } u+v-1 = max(I) \\ & \text{there is nothing to check.} \end{cases}$$

$\square_{Lemma\ 2}$

Lemma 3 *Let us assume there exists a protocol $\mathcal{AP}$ and a protocol history $h = (e_i)_{i \in \mathbb{N}}$ in which at least one correct process does not terminate. That is $\forall k \quad \neg \text{TERMINATED}^k$. Moreover let us assume that $\forall l \, \exists k \quad \text{PERFECT}_l^k$. Then either $\exists k \quad \text{TERMINATED}^k$, or $\exists l \, \forall k \quad \neg \text{PERFECT}_l^k$. That is either the protocol terminates, or the perfectness of the failure detector is length bounded.*

Proof Let us assume that (1) $\forall k \quad \neg \text{TERMINATED}^k$ and (2) $\forall l \, \exists k \quad \text{PERFECT}_l^k$. Since $\mathcal{AP}$ is $\diamond\mathcal{P}$-TERMINATING, (2) implies, for $l = l_0$ that there exists k such that PERFECT_l^k, which in turn, implies that TERMINATED^{k+l}. That is (1)$\wedge$(2)$\Rightarrow$ *false*. One can deduce that either (1), or $\neg$(2) is satisfied. That is either the protocol terminates, or the perfectness of the failure detector is length bounded. $\square_{Lemma\ 3}$

Lemma 4 *Let $(x_i)_{(i \in [1,n])}$, be n positive reals. Let $x = \sum_{i \in [1,n]} x_i$. Then $\sum_{i \in [1,n]} x_i^2 \geq \frac{x^2}{n}$.*

Proof The proof is by recurrence on n, and relies on the fact that the function $f : x \to x^2$ is convex. That is $\forall \lambda \in [0,1], \quad \forall x, y \in \mathbb{R}, \quad f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda) f(y)$. The base case is $n = 2$, where one applies the convexity inequality for $\lambda = 1/2$: $\frac{x_1^2 + x_2^2}{2} \geq \left(\frac{x_1 + x_2}{2}\right)^2 = x/4$. Assume

that for all $m < n$, one has $\sum_{i\in[1,m]} x_i^2 \geq \frac{x^2}{m}$. One has:

$$\begin{aligned}\sum_{i\in[1,n]} x_i^2 &= \sum_{i\in[1,n-1]} x_i^2 + x_n^2, \\ &\geq \frac{(x-x_n)^2}{n-1} + x_n^2, \text{(by applying recursion assumption)} \\ &= \frac{x^2 - 2xx_n + nx_n^2}{n-1}.\end{aligned}$$

The previous polynome of the x_n variable, is always positive and reach its minimum for $x_n = x/n$. That is:

$$\begin{aligned}\sum_{i\in[1,n]} x_i^2 &\geq \frac{x^2 - 2x(x/n) + n(x/n)^2}{n-1}, \\ &\geq \frac{x}{n}.\end{aligned}$$

$\square_{Lemma\ 4}$

Proof (of Theorem 3) If all processes crash, the protocol terminates and hence, the theorem is satisfied. Let us assume that there exists at least one correct process. By lemma 3 the perfectness of the failure detector is length bounded. In that case, we show that there exists an overlapping sequence with infinite average round trip delay.

Since there is a finite number $n = |\Pi|$ of processes in the system, all the processes that are faulty will eventually crash. Thus there exists an integer t_0 such that after index t_0 all processes that take steps in h are correct. This implies that after t_0, all messages that are sent, are sent by correct processes. Furthermore, there exists an index $t_1 > t_0$ such that all messages that are delivered after t_1 have been sent by a correct process (since only a finite number of messages had been sent before t_0 and they have been either received by t_1 or will never be received).

Recall that any event e is triggered by a unique message. Let us denote this message by $m(e)$. Furthermore recall that we are using a $\mathcal{FD}_L$ failure detector that simulates an eager failure detector. By theorem 1 one knows that failure detectors of class $\mathcal{FD}_L$ are complete, that is, crashed processes are eventually suspected by correct processes. That means that there exists an index $t_2 > t_1$, such that all faulty processes are suspected by all correct processes. Since all faulty processes have crashed, the only reason for the predicate $\textsc{Perfect}_l^{k\geq t_2}$ not to be satisfied is a false suspicion. That is, a correct process is erroneously suspected ((PA) property of the definition).

In the remainder, we use the previous results to build an infinite overlapping sequence with an average round trip delay that is unbounded. By assumption 1, the probability that such a sequence exists is equal to 0. That implies that a valid protocol history h with a bounded length of perfect failure detection can only exist with probability 0. Thus in turn, this allows to conclude that $\Diamond\mathcal{P}$-terminating protocols terminate with probability 1.

We build recursively an overlapping subsequence of history h, starting at event e_{t_2+1}. Let us define $k_0 = t_2 + 1$, and assume that we have build an overlapping sequence S_j up to k_j. We show how to build an extension of S_j, called S_{j+1}, that is still overlapping. By assumption, one knows that predicate $\textsc{Perfect}_l^{k_j}$ is not satisfied. More precisely, we know that there exists a smallest index l_j with $k_j \leq l_j \leq k_j + l - 1$ and at which occurs a false suspicion. Let us remark that the sequence $(e_i)_{k_j\leq i\leq l_j}$ is overlapping. Indeed e_i is triggered by $m(e_i)$, whose round trip delay is finite (because $m(e_i)$ has been sent by a correct process, and its acknowledgment will eventually be received). Moreover since there is no event between e_i and e_{i-1}, at least $m(e_i)$ is in transit at the time $m(e_{i-1})$ is received.

Since there is a false suspicion at event $e_{l_j} = (p, t, \_, \_, m(e_{l_j}))$, there must exist a message m_{l_j} sent by a process p to process q and p has queries the status of q at event e_{l_j} and p has not received the acknowledgment of m_{l_j}. Two cases have to be considered, either m_{l_j} has not yet been received by q at time t, or m_{l_j} has already been received by q at time t.

Let us consider the first case, where m_{l_j} is received after t. There must exist an event e_{r_j} triggered by m_{l_j}, that is $e_{r_j} = (q, t' > t, \_, \_, m_{l_j})$. Since m_{l_j} is received after t, and sent before t, that means that m_{l_j} is in transit by the time $m(e_{l_j})$ is received. Hence, e_{r_j} and e_{l_j} are overlapping. Moreover, there must exist an event $e_{k_{j+1}}$ triggered by the receipt of $ack(m_{l_j})$ at process p, that is $e_{k_{j+1}} = (p, t'' > t', \_, \_, ack(m_{l_j}))$. Since m_{l_j} is received at time t', when $ack(m_{l_j})$ is sent, m_{l_j} and $ack(m_{l_j})$ are overlapping. Hence, the sequence $s_j = (e_{k_j}, \ldots, e_{l_j}, e_{r_j}, e_{k_{j+1}})$ is overlapping. One can extend the sequence $S_j = (e_{k_i}, \ldots, e_{k_{i+1}})_{i \leq j}$ to $S_{j+1} = (e_{k_i}, \ldots, e_{k_{i+1}})_{i \leq j+1}$ by Lemma 1. That concludes the first case.

Let us consider the second case where m_{l_j} is received before t. There must exist an event e_{r_j} triggered by m_{l_j}, that is $e_{r_j} = (q, t' < t, \_, \_, m_{l_j})$. Message $ack(m_{l_j})$ is sent at time t' and thus before event e_{l_j}. Moreover, there must exist an event[3] $e_{k_{j+1}}$ triggered by the receipt of $ack(m_{l_j})$ at process p after t, that is $e_{k_{j+1}} = (p, t'' > t, \_, \_, ack(m_{l_j}))$. Thus $m(e_{l_j})$ is received at p while $ack(m_{l_j})$ is in transit. So events e_{l_j} and $e_{k_{j+1}}$ are overlapping. This allows to conclude that the sequence $S' = (e_{k_j}, \ldots, e_{l_j}, e_{k_{j+1}})$ is overlapping. Due to Lemma 1, the concatenation of S_j and S' is overlapping. Moreover, since m_{l_j} is received at time t' at which $ack(m_{l_j})$ is sent, m_{l_j} and $ack(m_{l_j})$ are overlapping. That implies that the sequence $S'' = (e_{r_j}, e_{k_{j+1}})$ is also overlapping. By Lemma 2 the sequence $S_{j+1} = sort(S_j + S', S'')$ is overlapping.

We have been able to extract from h an infinite subsequence of overlapping messages, where there is an unbounded number of false suspicions. Let A denote the set of pairs (i, r) such that p_i has wrongly suspected p_r an infinite number of times. This set is not empty. Indeed each of these suspicions occurs at a given process, since there is only a finite number of processes, there exists at least one process p_i at which occurs an infinite number of false suspicions. Furthermore, since there is a finite number of processes, one can deduce that process p_i has suspected infinitely often another process, say, p_r. Hence, (i, r) belongs to A. By definition of A, we can deduce that for each pair $(i, r) \in A$ process p_i has increased its variable $max\_rtd_i[r]$ an infinite number of times by at least g. Furthermore, one can show that there exists an index a such that in each extension $s_{k, k \geq a}$ of S_k, the false suspicion is due to p_i about p_j with $(i, j) \in A$.

Let us define the $MARTD$ operator on overlapping sequences of messages by:

$$MARTD(S) = \frac{\sum_{m \in S} RTD(m)}{|S|}.$$

Since $\mid S_k \mid \leq (k+1)(l+1)$, one has:

$$MARTD(S_k) \geq \frac{\sum_{m \in S_k} RTD(m)}{(k+1)(l+1)}.$$

Let us consider that every message in S_k has a zero round trip delay, except messages m_{l_k} send by p_i to p_j with $(i, j) \in A$. Let us denote $n_{i,j}(k)$ the number of times p_i has increased its variable $max\_rtd_i[j]$ after index a and before index k. Let us remark that for all $(i, j) \in A$ and for all $k \geq a$, $\sum_{(i,j) \in A} n_{i,j}(k) = k - a + 1$. Using Lemma 4, one can deduce that for all $(i, j) \in A$ and for all

[3] Otherwise there is no false suspicion.

$k \geq a$, $\sum_{(i,j)\in A} n_{i,j}(k)^2 \geq \frac{(k-a+1)^2}{|A|}$. Thus:

$$\begin{aligned}
\forall\, k \geq a \quad MARTD(S_k) &\geq \frac{\sum_{(i,j)\in A}\sum_{z=1}^{n_{i,j}(k)} zg}{(k+1)(l+1)}, \\
&\geq g\frac{\sum_{(i,j)\in A}(n_{i,j}(k)+1)n_{i,j}(k)/2}{(k+1)(l+1)}, \\
&\geq g\frac{\sum_{(i,j)\in A} n_{i,j}(k)^2 + n_{i,j}(k)}{2(k+1)(l+1)}, \\
&\geq g\frac{\mid A \mid + \sum_{(i,j)\in A} n_{i,j}(k)^2}{2(k+1)(l+1)}, \\
&\geq g\left(\frac{\mid A \mid}{(k+1)(l+1)} + \frac{(k-a+1)^2}{\mid A \mid (k+1)(l+1)}\right).
\end{aligned}$$

One can deduce that $MARTD(S) = \lim_{k\to+\infty} MARTD(S_k) = +\infty$. One knows that:

$$\forall\, c > 0 \quad P[MARTD(S) - E(MARTD(S)) \geq c] \leq \frac{1}{(V(MARTD(S))c)^2}$$

By assumption 1, one can deduce that $P[\forall B > 0 MARTD(S) - E(MARTD(S)) > B] = 0$. That is, our initial assumption occurs with probability 0. In other words AF terminates with probability 1.

$\square_{Theorem\ 3}$

6 Experimental Results

This section describes performance measurements of an implementation of $\mathcal{FD}_L$ in three different network settings.

Experiment Program This program consisted of a process pair (p, q) where each of the two processes is made up of an infinite lop where it queries the status of the other process. Between two queries, each process waits for a minimum amount of time, namely, the currently experienced maximum round-trip delay, i.e., $max\_rtd_p[q]$ for process p.

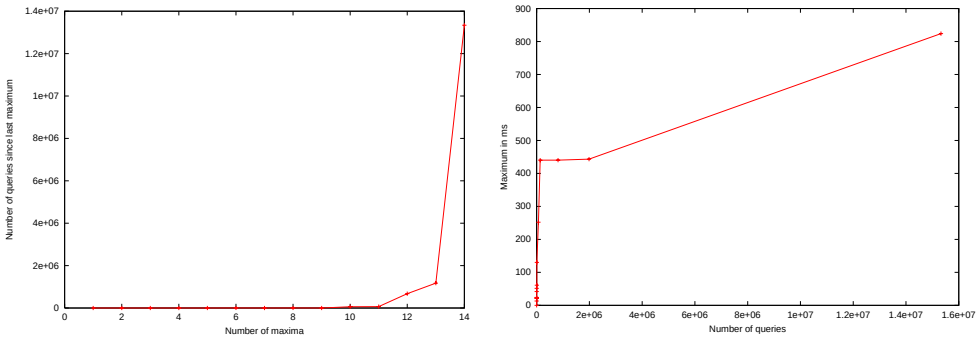
Measured values The following values have been measured for each process p:
- The number of queries between two maxima, i.e., the number of queries of p between updates of the variable $max\_rtd_p[q]$, and
- The value of $max\_rtd_p[q]$ with respect to the number of queries.

Context of the measures The measurements have been performed in three different types of system.

- First, we measured the performance for p and q running on the same computer with the Linux system (see Figures 4(a) and 4(b)). The total number of queries was 19 270 000 and the maximum round-trip delay was $0.824s$.
- The second measurement was performed on two computers connected via an asymmetric cable modem and one of these computer was connected via a wireless 802.11b Ethernet interface to the cable modem (see Figures 5(a) and 5(b)). This setup shows a substantially higher

maximum. Former measurements indicate that this is caused by the very unreliable and slow phone uplink of the cable modem. The total number of queries was 4 320 000 and the maximum experienced round-trip delay was 185.890s.

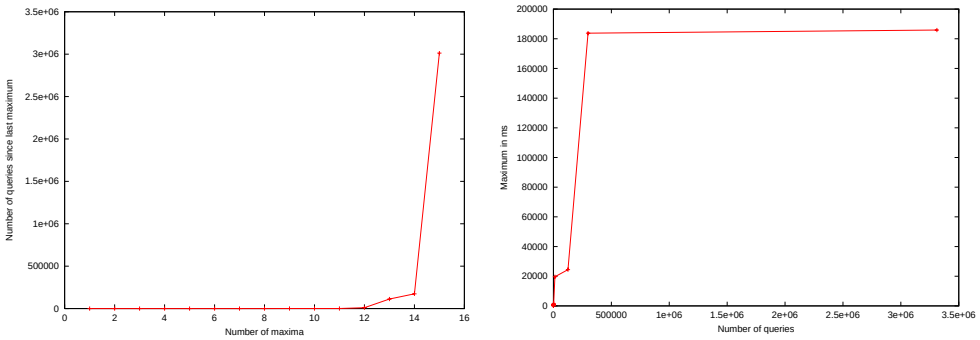
- The final measurement was performed on two computers connected via the Internet (see Figures 6(a) and 6(b)). One computer was located in San Diego, US while the other was located in Rennes, France. The total number of queries was 2 050 000 and the maximum experienced round-trip delay was 31.681s.



(a) Number of queries between consecutive maxima

(b) Number of queries between consecutive maxima

Figure 4: Local processes

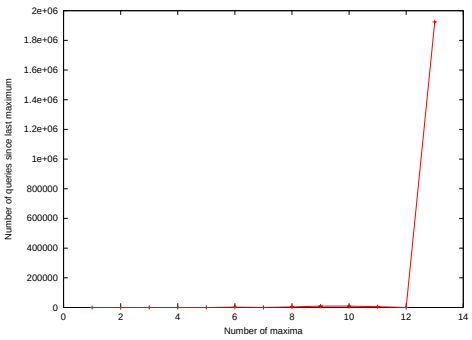


(a) Number of queries between consecutive maxima

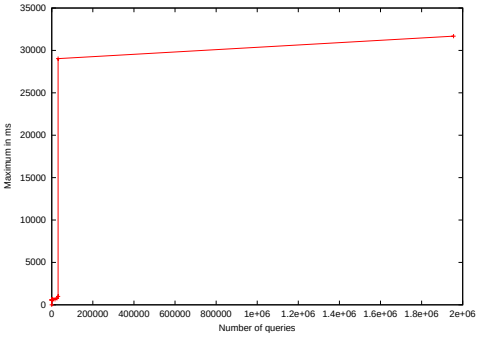
(b) Number of queries between consecutive maxima

Figure 5: Processes connected via a cable modem

Lesson learned The lesson learned from the experiments can be extracted from the Figures 7(a) and 7(b). These figures show that there is an exponential increase in queries between two consecutive maxima. Interestingly, even though the underlying network technologies and experienced



(a) Number of queries between consecutive maxima

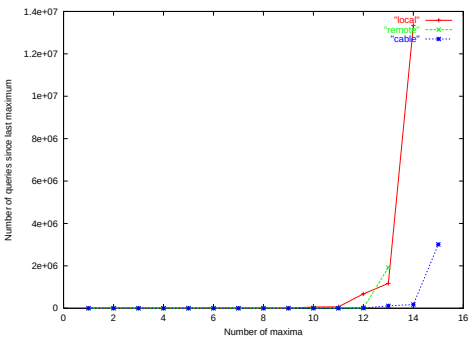


(b) Number of queries between consecutive maxima

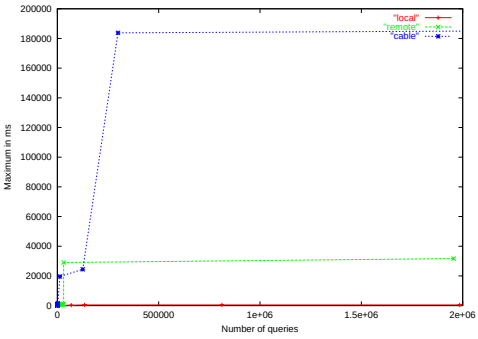
Figure 6: Remote processes connected via the internet

maxima are quite different, the distances between consecutive maxima are comparable (Figure 7(a)).

Hence, in order to reduce the number of wrong suspicions, it makes sense that a practical implementation keeps track of $max\_rtd_p[q]$ between executions. In this way, one could avoid almost all wrong suspicions.



(a) Number of queries between consecutive maxima



(b) Number of queries between consecutive maxima

Figure 7: Summary for three different network technologies

7 Conclusion

This paper presented a relatively simple protocol that allows a process to "monitor" another process, and consequently to detect its crash. This protocol enjoys the nice property to rely as much as possible on application messages to do this monitoring. Differently from previous process crash detection protocols, it uses control messages only when no application messages is sent by the monitoring process to the observed process.

It has been shown that the proposed protocol has noteworthy features. When the underlying system satisfies the partial synchrony assumption, it actually implements an eventually perfect failure detector (i.e., a failure detector of the class usually denoted $\diamond\mathcal{P}$). Moreover, if the upper layer application terminates correctly when the failure detector it uses belongs to $\diamond\mathcal{P}$, then, when run with the proposed protocol, it also terminates correctly. These properties make the protocol attractive: it is inexpensive, implementable, and powerful.

The paper also described performance measurements of several implementations of the protocol. These measurements show that the number of wrong suspicions can be reduced by requiring each process p to keep track $max\_rtd_p[q]$ between executions.

Acknowledgments

The authors are grateful to Michel Hurfin for interesting discussions on the failure detection problem.

References

[1] Beauquier J. and Kekkonen-Moneta S., Fault-Tolerance and Self-Stabilization: Impossibility Results and Solutions Using Self-Stabilizing Failure Detectors. *Int. Journal of Systems Science*, 28(11):1177-1187, 1997.

[2] Chandra T. and Toueg S., Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225-267, 1996.

[3] Chandra T., Hadzilacos V. and Toueg S., The Weakest Failure Detector for Solving Consensus. *Journal of the ACM*, 43(4):685–722, 1996.

[4] Cristian F., Probabilistic Clock Synchronization. *Distributed Computing*, 3:146-158, 1989.

[5] Dwork C., Lynch N. and Stockmeyer L., Consensus in the Presence of Partial Synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

[6] Fischer M.J., Lynch N. and Paterson M.S., Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374–382,1985.

[7] Gafni E., Round-by-Round Failure Detectors: Unifying Synchrony and Asynchrony. *Proc. 17th ACM Symposium on Principles of Distributed Computing (PODC'98)*, ACM Press, pp. 143-152, Puerto Vallarta (Mexico), 1998.

[8] Garg V.K. and Mitchell J.R., Implementable Failure Detectors in Asynchronous Systems. *Proc. 18th Int. Conference on Foundations of Software Technology and Theoretical Computer Science (FST&TCS'98)*, Springer-Verlag LNCS #1530, pp.158-169, New-Delhi (India), 1998.

[9] Hélary J.-M., Hurfin M., Mostefaoui A., Raynal M. and Tronel F., Computing Global Functions in Asynchronous Distributed Systems with Process Crashes. *IEEE Transactions on Parallel and Distributed Systems*, 10(9), 2000.

[10] Larrea M., Arèvalo S. and Fernández A., Efficient Algorithms to Implement Unreliable Failure Detectors in Partially Synchronous Systems. *Proc. 13th Symposium on Distributed Computing (DISC'99)*, Springer-Verlag LNCS #1693, pp.34-48, Bratislava (Slovaquia), 1999.

[11] Larrea M., Fernández A. and Arèvalo S., Optimal Implementation of the Weakest Failure Detector for Solving Consensus. *Brief Announcement, Proc. 19th ACM Symposium on Principles of Distributed Computing (PODC'00)*, ACM Press, p. 334, Portland (OR, USA), 2000.

[12] Lee I. and Davidson B., Adding Time to Synchronous Process Communication. *IEEE Transactions on Software Engineering*, 36(8):941-948, 1987.

[13] Mostéfaoui A. and Raynal M., Solving Consensus Using Chandra-Toueg's Unreliable Failure Detectors: a General Quorum-Based Approach. *Proc. 13th Int. Symposium on Distributed Computing (DISC'99)*, Springer-Verlag LNCS #1693, pp. 49-63, Bratislava (Slovaquia), 1999.

[14] Raynal M. and Tronel F., Group Membership Failure Detection: a Simple Protocol and its Probabilistic Analysis. *Distributed Systems Engineering Journal*, 6(3):95-102, 1999.