On the Fly Estimation of the Processes that Are Alive in an Asynchronous Message-Passing System

Achour Mostefaoui, Michel Raynal, and Gilles Tredan

Abstract—It is well known that in an asynchronous system where processes are prone to crash, it is impossible to design a protocol that provides each process with the set of processes that are currently alive. Basically, this comes from the fact that it is impossible to distinguish a crashed process from a process that is very slow or with which communications are very slow. Nevertheless, designing protocols that provide the processes with good approximations of the set of processes that are currently alive remains a real challenge in fault-tolerant-distributed computing. This paper proposes such a protocol, plus a second protocol that allows to cope with heterogeneous communication networks. These protocols consider a realistic computation model where the processes are provided with nonsynchronized local clocks and a function $\alpha()$ that takes a local duration Δ as a parameter, and returns an integer that is an estimate of the number of processes that could have crashed during that duration Δ. A simulation-based experimental evaluation of the proposed protocols is also presented. These experiments show that the protocols are practically relevant.

Index Terms—Approximation protocol, asynchronous system, coverage assumption, crash failure, crash detection, fault-tolerance, message passing, nonsynchronized local clocks.

1 Introduction

Detecting Process Crashes. A synchronous message-passing system is a system in which there are bounds on message transfer delay and processing time. Moreover, these bounds are known by the processes that can consequently use them in their computation. Considering a synchronous message-passing system prone to process crashes (without recovery), a main problem in fault-tolerant-distributed computing consists in designing a protocol that allows a process to determine an estimate of the number of crashes. An easy protocol suited to synchronous systems is as follows: Let us assume, without loss of generality, that the processing of a message takes no time (the processing time of a message can be included in its transfer delay). Consider that a process answers an inquiry message in return. To compute an estimate of the current number of crashed processes, a process broadcasts an inquiry message, sets a timer to the maximal round-trip delay, and waits until the timer expires. It can then safely conclude that all the processes from which it has not received an answer before the timer expiration have crashed. Moreover, this estimation is an underestimate as a process can crash just after having sent its answer.

Conversely, an asynchronous message-passing system is characterized by the fact that there is no assumption on message transfer delay (except the fact that any delay is finite) [1], [13]. It follows that there is no notion of "maximal round-trip delay" in pure asynchronous systems. Consequently, even if a process is allowed to use timers, there is no way for it to safely detect process crashes, whatever the time-out values it uses. The impossibility to distinguish a crashed process from a process with which communication is very slow is one of the main difficulties one has to face when designing distributed fault-tolerant services. The most famous related result is the impossibility to solve the consensus problem in asynchronous systems prone to even a single-process crash failure [8].

The classical asynchronous distributed system model. Let n be the number of processes that the system is made up of (we consider static systems). A classical way to address the previous drawback consists in augmenting the asynchronous system model with an additional parameter, usually denoted t ($t < n$), that is assumed to be an upper bound on the number of processes that can crash. This parameter t can be seen as a guess on the future behavior of the system.

Let us consider, in that model, the simple problem of a process p that wants to obtain data from "as many processes as possible." To that end, p broadcasts a query. Classically, the model parameter t is used to define a logical deadline after which the querying process stops waiting for responses: it waits until it has received $n - t$ responses (without using any timer).

Let us define the *response quality* associated with a given query as the number of responses (to that query) that are received. The classical asynchronous system model has two weaknesses when we are interested in the response quality criterion. Let f be the actual number of process crashes.

- If $f < t$, as the querying process waits only for $n - t$ responses, it is missing $t - f$ responses. This can be

• *The authors are with IRISA, Université de Rennes 1, Campus de Beaulieu, 35042 Rennes Cedex, France. E-mail: {achour, raynal, gtredan}@irisa.fr.*

Manuscript received 21 July 2008; accepted 16 Dec. 2008; published online 7 Jan. 2009.
Recommended for acceptance by M. Singhal.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2008-07-0273.
Digital Object Identifier no. 10.1109/TPDS.2009.12.

particularly penalizing when f is small, as the more responses the querying process obtains, the better it is. More precisely, the response quality is always $n - t$ (whatever the actual number of process crashes), while it could be up to $n - f$.

- If $f > t$, the querying process blocks forever. This is because, after it has received $n - f$ responses, the querying process keeps on waiting for the $f - t$ missing responses that will never be sent.

In the first case, the response quality can be severely reduced. In the second case, while the model parameter t is assumed to be an upper bound on the number of process crashes in any execution, it appears that t is not such an upper bound in some executions. Of course, a greater value of t could have been chosen, but in that case, the response quality could become very low in the executions where f is much smaller than t. Finding a good approximation of the current number of crashed processes is consequently one of the challenges one has to take up when one wants to ensure a good response quality while preventing process permanent blocking.

Related Work. The problem posed by process crashes has received a lot of attention in distributed agreement problems. The impossibility to solve such problems in asynchronous systems prone to process crashes has been proved in [8]. Ways to circumvent this impossibility are presented in [3] (failure detector approach; see [17] for a survey of this approach), [5] (additional synchrony), [2], and [16] (randomized protocols).

A failure detection service suited to large-scale systems and based on a gossip service is proposed in [19]. Group membership failure detection is investigated in [18]. Adaptive failure detection is considered in [7]. Failure detection in the timed asynchronous system model [4] is addressed in [6]. Krishnamurthy et al. [11] studies the degradation of the quality of service due to uncertain variations in (load and) unanticipated failures. Hayashibara et al. [9] present a new approach that associates a suspicion level (on a continuous scale) with each process, and dynamically adjusts to network conditions the scale on which the suspicion level is expressed.

Content of the Paper. The paper is on the determination on the fly of the processes that are alive in an asynchronous message-passing system. It has several contributions.

- The first contribution is a new model for asynchronous message-passing systems. As we have seen, the classical parameter t is an *assumption* that can be satisfied or not in a given execution. The proposed model replaces that assumption by another assumption on the maximum number of processes that could crash during a given time duration.

  More precisely, the model assumes that the processes are provided with nonsynchronized local clocks and can invoke a function (denoted as $\alpha()$) that takes a duration as a parameter. $\alpha(\Delta)$ returns to the invoking process, an integer that is an estimate of the number of processes that can crash during Δ units of time.
- An $\alpha()$-based distributed protocol is presented and proved to be correct. That protocol provides each process with an estimate of the processes that are currently alive. The proposed protocol is based on a simple query-response mechanism and the local clock of the querying process (to make the presentation easier, a global clock-based protocol is first presented). That protocol has the following noteworthy properties: It imposes no constraint on the number of processes that can crash, and always terminates. So, it does not suffer the previous limitation inherent to the explicit t-based model.
- A simulation study is presented that evaluates the quality of the set currently output at each process. It appears that this quality is pretty good as an alive process does not remain suspected for a long time and a crashed process is quickly suspected.
- A protocol is presented that allows heterogeneous communication to be taken into account. In such systems, a high degree of clustering can increase the number of rounds needed to converge. The approach proposed to overcome this problem consists in requiring a process to wait before proceeding to the next communication round. The duration of this additional waiting period for the round $r + 2$ is a function that takes into account the local arrival dates of late messages (messages that arrive at round $r + 1$ while they should have arrived at round r).

Roadmap. The paper is made up of seven sections. Section 2 presents the computation model. Then, Section 3 introduces two distributed protocols that provide each process p with a set containing the processes that p can consider as the processes currently alive. The presentation of these protocols is incremental: The first protocol is based on a global clock that all the processes can read. The second protocol shows that that global clock is not mandatory and can be replaced by nonsynchronized local clocks. Then, Section 4 presents a simulation study of the local clock-based protocol. It shows that the protocol is both meaningful and fast. "Meaningful" means that the probability to suspect an alive process is very small; "fast" means that an alive process that is currently suspected becomes very quickly nonsuspected. Section 5 proposes an extended computation model including both the parameter t and $\alpha()$. Section 6 proposes a protocol that allows to deal with heterogeneous communication networks and presents a simulation study. Finally, Section 7 concludes the paper.

2 System Model

Asynchronous System. The system is made up of a set $\Pi = \{p_1, \ldots, p_n\}$ of n processes (nodes) that communicate by exchanging messages. Each process proceeds at its own speed and, though each message takes a finite time for going from its sender to its receiver, there is no bound on message transfer delays. This means that, on both the process side and message side, the system is asynchronous.

Failure model. The underlying network is assumed to be reliable in the sense that no message can be lost, duplicated, or corrupted. Moreover, if a message is received, it has previously been sent by a process. While the asynchronous communication network is reliable, processes are not. A process can crash (i.e., it definitely stops executing operations). Given a system execution, a process that crashes is said

```
operation estimate():
(01)  start_time_i ← global_clock();
(02)  broadcast QUERY();
(03)  wait until (|est_i.set| − β)) corresponding RESPONSE(rec_from_j) have been received
(04)        where β = α(global_clock() − est_i.date) is continuously evaluated;
            % If any, when they arrive, the other corresponding response messages are discarded %
(05)  rec_from_i.date ← start_time_i;
(06)  rec_from_i.set  ← the set processes from which p_i has received RESPONSE() at line 03;
(07)  est_i.date ← min over the rec_from_j.date received at line 03;
(08)  est_i.set  ← ⋃ of the rec_from_j.set received at line 03;
(09)  return(est_i.set)

background task T:
  when QUERY() is received from p_j do send RESPONSE(rec_from_i) to p_j end_do
```

Fig. 1. The estimate() operation (version based on a global clock).

to be *faulty* (in that execution). A process that does not crash is said to be *correct* (in the corresponding execution). A process is *alive* until it (possibly) crashes.

Broadcast Operation. The processes are provided with a BROADCAST (m) operation, where m is a message. Such an operation is not atomic. It can be seen as a shortcut for: "**for each** $p_j \in \Pi$ **do** send (m) to p_j **end do**."

This means that if the sender does not crash while executing this operation, the message m is sent to all the processes, including its sender. If the sender crashes, the message m is sent to an arbitrary subset of the processes.

Local Clock. Each process is provided with a local clock. It obtains the current local date by invoking the operation local_clock(). The local clocks of the processes are not synchronized: They can have different values at the same real-time instant τ.

We assume that the local clocks are drift-free. (Considering local clocks that have a drift upper and lower bounds is possible. We do not consider it as it would only add a syntactic burden to our presentation.) A local clock is used only to measure the time duration that elapses during two local events. It is assumed that the grain of a local clock is such that the clock increases between consecutive local events.

$\alpha()$ **Function.** As we have seen in the introduction, in order to design useful nontrivial protocols despite process crash occurrences, a system model has to include some assumption on the system behavior. Such an assumption is a guess that, when satisfied by the system, allows the design of correct and nonblocking protocols (as we have seen in Section 1, the classical parameter t is such an assumption).

The parameter t is static in the sense that it is defined once and forever. So, instead of t, we propose a system model that provides the processes with an operation, denoted as $\alpha()$ that, taking a duration Δ as a parameter, returns to the invoking process an integer belonging to $[0..n-1]$. That integer is an estimate of the number of processes that could crash during Δ time units. As can be seen, the function $\alpha()$ provides a guess on the system behavior, but this guess is more dynamic than t. To be useful, the function $\alpha()$ has to satisfy the following properties.

- $\Delta_1 \geq \Delta_2 \Rightarrow \alpha(\Delta_1) \geq \alpha(\Delta_2)$ (nondecreasing).
- It eventually increases according to the duration (the more time elapses, the more processes can crash). More formally: $\forall \Delta_1$ such that $\alpha(\Delta_1) < n-1$, $\exists \Delta_2 > \Delta_1$ such that $\alpha(\Delta_2) > \alpha(\Delta_1)$.

From a practical point of view, $\alpha()$ can be defined from observations of previous system runs, these runs providing a realistic value for the maximum number of processes that crash per time unit.

Remark. When we consider the particular case where the function $\alpha()$ always returns $n-1$ (whatever the value of Δ), we obtain the particular case of the classical asynchronous model where, at any time, the only assumption a process can rely on is that at most $t = n-1$ processes have crashed.

3 Computing Approximations of the Set of Alive Processes

This section presents a protocol that provides each process with the set of the processes that are deemed to be alive. An estimate of the set of crashed processes can easily be computed by subtracting this set from Π (the whole set of processes). To make it easier to understand, as mentioned in Section 1, the protocol is presented incrementally in two steps. We first assume that the processes have access to a common global clock. Then, that global clock is approximated with the nonsynchronized local clocks. Without loss of generality, we assume that local processing by the processes takes no time, only message transfer takes (arbitrary) time.

3.1 A Global Clock-Based Protocol

So, let us assume that the system provides the processes with a common clock that they can read by invoking the operation global_clock(). Each process regularly executes the operation estimate() described in Fig. 1, which provides it with the set of processes deemed to be alive. This set is returned at line 09.

Local variables. Each process p_i manages three local variables.

- The variable est_i is composed of two fields: $est_i.set$ and $est_i.date$. The field $est_i.set$ represents p_i's current estimate of the processes that are currently alive. As we will see, all the processes belonging to this set

were alive at time $\tau = est_i.date$ (when $est_i.date > 0$). (Initially, $est_i.set = \Pi$ and $est_i.date = 0$.)[1]

- The variable rec_from_i is also composed of two fields: $rec\_from_i.set$ is the last set of processes from which p_i has received response messages; $rec\_from_i.date$ is a conservative date indicating that the processes of $rec\_from_i.set$ were not crashed at time $rec\_from_i.date$.
- $start\_time_i$ is an auxiliary variable that contains the last date at which p_i sent a query message.

Process behavior. As indicated, until it possibly crashes, each process p_i repeatedly executes the operation estimate(). That operation consists of a query/response mechanism as introduced in [14]: p_i issues a query (line 02) and waits for corresponding responses (line 03).[2] (The responses received after p_i stops waiting are discarded.) Then, p_i computes the new values of rec_from_i and est_i as follows.

- $rec\_from_i.set$ is the set of processes from which p_i, during its waiting period (lines 03-04), has received responses to its last query. Let us observe that, whatever the round-trip delays associated with these query/responses, all the processes that sent a response were alive when p_i issued the query, i.e., at time $start\_time_i$. Consequently, $rec\_from_i.date$ is set to $start\_time_i$ (lines 05).
- When a response to a query (issued by a process p_i) is sent back to p_i by a process p_j, that response carries the current value of the local variable rec_from_j (see the background task).

  The union of the sets $rec\_from_j.set$ received by p_i are used to define the new value of $est_i.set$ (line 08), these processes are the processes deemed alive by p_i. Moreover, as the processes in $rec\_from_j.set$ were alive at time $\tau_j = rec\_from_j.date$, we can conclude that the processes in $est_i.set$ were alive at time $\min(\tau_{j_1}, \ldots, \tau_j, \ldots, \tau_{j_x})$ (where each τ_j corresponds to a response received by p_i during its waiting period); $est_i.date$ is accordingly set to that date (line 07).

It now remains to specify, for each query, how many responses a process p_i has to wait for (lines 03 and 04). This is where the $\alpha()$ function provided by the model comes into play. Until it stops waiting, p_i repeatedly evaluates $\beta = \alpha$ (global_clock() $- est_i.date$) (line 04). The current value of β is an approximation of the number of processes that could have crashed since the date $\tau = est_i.set$ up to now. Then, p_i waits until it has received response messages from $|est_i.set| - \beta$ processes, i.e., the set of processes it considers alive at time $\tau = est_i.set$, minus the processes that could have crashed since that time (line 03). (As already indicated, the responses that arrive too late are discarded.)

Properties. It is important to see that a set $est_i.set$ can decrease or increase according to the values of the sets $rec\_from_j.set$ received by its process p_i.

1. As shown in Theorem 2, it is actually possible to initialize $est_i.set$ to any subset of Π. The simulations described in Section 4 consider that initially, each process falsely suspects some part of the other processes (expressed as a percentage varying from 0 percent up to 95 percent).

2. Each query can carry an identity, and the corresponding responses can be identified with the same identity. To not overload the presentation, these query identities are left implicit.

The following theorems state the properties provided by the protocol. Theorems 1 and 2 define the safety property ensured by the protocol. More precisely, Theorem 1 states that every crash is eventually detected, while Theorem 2 gives its meaning to $est_i.date$ (namely, the processes returned by an invocation were alive at time $est_i.date$). Theorem 3 addresses the liveness of the protocol.

Theorem 1. *Let us consider an execution in which p_k is a faulty process and p_i is a correct process. There a time after which p_k does not belong to $est_i.set$.*

Proof. Let τ be a time after which no process receives responses from p_k (as p_k crashes, it sends only a finite number of response messages, and consequently the time τ does exist). This means that from τ, no process p_j includes p_k in its set $rec\_from_j.set$. It is possible that at τ, there are response messages that are in transit and carry a set including p_k; if it is the case, the number of such messages is finite. Let $\tau' \geq \tau$ be a time after which no process receives a response carrying a $rec\_from.set$ set including p_k (due to the previous observation and the fact that any message takes a finite time, τ' does exist). It follows from lines 03 and 08 that, after τ', no process p_i insert p_k in its set $est_i.set$. □

Theorem 2. *Let us consider an invocation estimate() issued by a process p_i. None of the processes in the $est_i.set$ returned as that invocation was crashed at time $\tau = est_i.date$.*

Proof. Let us first observe that any process placed in the set $rec\_from_j.set$ by a process p_j was alive when p_j issued the corresponding query (otherwise that process could not send back a response to that query). The processes that are placed in $rec\_from_j.set$ were consequently alive at the time $\tau_j = rec\_from_j.date$ (the date at which p_j issued the query).

Let us now consider a process p_i that computes $est_i.date$ and $est_i.set$ at lines 07 and line 08, respectively. The theorem follows from the facts that: 1) $est_i.set$ is the union of the $rec\_from_j.set$ just received, and 2) $est_i.date$ is the smallest of the associated $rec\_from_j.date$ dates. □

Theorem 3. *Every invocation of estimate() by a correct process terminates.*

Proof. The only statement where a correct process p_i can block forever is the **wait until** statement (lines 03 and 04). So, let us assume by contradiction that p_i blocks forever in this wait statement. Due to line 04, it continuously computes a new value for β. As $est_i.date$ does not change between successive computations of β, and the values returned by the successive invocations global_clock() always eventually increase, it follows from the properties of $\alpha()$ that the local predicate $|est_i.set| - \beta \leq 1$ eventually becomes true. As the links are reliable, p_i receives at least its response to its own query. Consequently, there is a time after which $|est_i.set| - \beta \leq 1$ is true and p_i has received its own response. When this occurs, p_i stops waiting, contradicting the initial assumption. □

3.2 A Local Clock-Based Protocol

This section adapts the previous protocol to a setting without a global clock, each process being provided only with a local clock. As the local clocks are not synchronized,

```
operation estimate():
(01)  broadcast QUERY();
(02)  wait until (|est_i.set| − β)) corresponding RESPONSE(rec_from_j, local_date_j, helping_date_j[i])
(03)        have been received where β = α(local_clock() − est_i.date) is continuously evaluated;
            % If any, when they arrive, the other corresponding response messages are discarded %
(04)  rec_from_i ← the set of processes from which p_i has received RESPONSE() at line 02;
(05)  for each p_j ∈ rec_from_i do helping_date_i[j] ← last_date_i[j];
(06)                               last_date_i[j]    ← local_date_j end do;
(07)  est_i.date ← min of the helping_date_j[i] received at line 02;
(08)  est_i.set  ← ∪ of the rec_from_j sets received at line 02;
(09)  return (est_i.set)

background task T: when QUERY() is received from p_j
                   do send RESPONSE(rec_from_i, local_clock(), helping_date_i[j]) to p_j end_do
```

Fig. 2. The estimate() operation (version based on nonsynchronized local clocks).

each clock is a "purely" local object (which means that the value of a given clock is meaningless outside its process).

The problem consists, for each process p_i, in associating a local date τ_i with each set $est_i.set$, such that τ_i is as recent as possible, and all the processes that belong to $est_i.set$ were alive at time τ_i (assuming an external observer that uses the local clock of p_i to timestamp all the events that occur in the system). As soon as such a time value is determined, p_i can use it to compute an approximation of the number of processes that can have crashed since the last computation of $est_i.set$ (as done at lines 03 and 04 of the global time-based protocol described in Fig. 1).

Local Variables. To attain the previous goal, each process is provided with some of the previous data structures plus new ones.

- est_i: This local variable is the same as the previous. It has two fields $est_i.set$ and $est_i.date$ with the same meaning. The only difference is that now $est_i.date$ refers to a local date defined from the local clock of p_i.
- rec_from_i: This local variable is now a simple set whose meaning is the same as $rec\_from_i.set$ in the previous protocol.
- Each process maintains two additional local arrays, denoted $helping\_date_i[1..n]$ and $last\_date_i[1..n]$. Their meaning is the following: When a process p_j returns a response to a query issued by p_i, it sends its current local time value (see the background task of Fig. 2). When it receives that time value (that is meaningful only for p_j), p_i stores it as $last\_date_i[j]$ (line 05). In that way, p_i is able to indicate to p_j the date (measured with p_j's local clock) at which p_j sent its last response to p_i.

  Unfortunately (as we will see in Theorem 4), this is not sufficient to guarantee the property stated above relating $est_i.set$ and $est_i.date$ (all the processes of $est_i.set$ were alive at $\tau_i = est_i.date$). We need to send back to p_j not the last date, but the previous one. That date is kept by p_i in $helping\_date_i[j]$.

Process Behavior. The behavior of p_i is described in Fig. 2. It is nearly the same as the behavior defined for the global time-based protocol. When p_i sends a response message to p_j, it sends the current value of the set rec_from_i, the current value of its local clock (to be helped by p_j), and the current value of $helping\_date_i[j]$ to help p_j in its duration computation.

When it receives a value $helping\_date_j[i]$ from a process p_j (line 02), p_i uses it to compute the date $est_i.date$ it associates with the set $est_i.set$ (line 07).

Properties. Theorems 1, 2, and 3 remain true when we consider the local clock-based protocol described in Fig 2. While the proofs of Theorem 1 and 3 are nearly the same, this is no longer true for Theorem 2. So, here we provide only a proof suited to the nonsynchronized local clock model for Theorem 2.

Theorem 4. *Let us consider an invocation* estimate() *issued by a process p_i as described in Fig. 2. None of the processes in the $est_i.set$ returned by that invocation was crashed at time $\tau = est_i.date$.*

Proof. To prove the theorem, we consider a process p_k that belongs to a set rec_from_j that p_i uses to define the new value of $est_i.set$ (line 08). We show that p_k was alive at p_i's local time $est_i.date$.

The corresponding situation is depicted in Fig. 3 where two queries issued by p_j are described such that the two corresponding response messages sent by p_i are processed by p_j. Let τ_i^1 and τ_i^2 be the current values of p_i's local clock when that process sent the corresponding response messages.

Due to the protocol, we have the following after p_j has processed the message $response_i^2$ from p_i and the message $response_k^2$ from p_k:

- $helping\_date_j[i] = \tau_i^1$ and $last\_date_j[i] = \tau_i^2$ (lines 05 and 06 executed by p_j).
- $p_k \in rec\_from_j$ (line 04 executed by p_j).

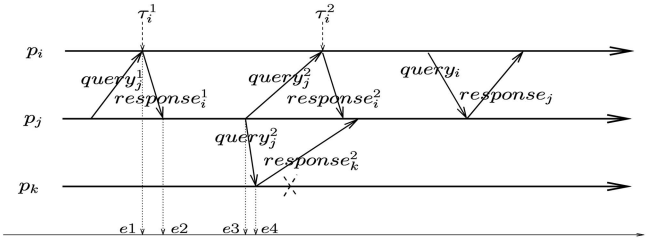

Fig. 3. Using the *happened before* relation.

Let us first observe that p_k can crash just after sending the message $response_k^2$ (this is indicated with a dotted cross in the figure). Such a crash might happen before p_i receives the $query_j^2$ message, which means that it is possible the p_k crashes before p_i's local clock becomes equal to τ_i^2. We conclude from that observation that when it receives from p_j the message $response_j$ (carrying rec_from_j such that $p_k \in rec\_from_j$), p_i cannot conclude that p_k was alive when its local time was τ_i^2.

As indicated by the protocol, the message $response_j$ sent by p_j carries $helping\_date_j[i] = \tau_i^1$, and accordingly, p_i uses that local date when it computes $est_i.date$ (line 07). We then have $est_i.date \leq \tau_i^1$. We show that p_k was alive when p_i's local clock was equal to τ_i^1. This follows from the following observation based on Lamport's *happened before* relation [12].

- p_j issues its second query ($query_j^2$) after it has processed the $response_i^1$ message from p_i.
- p_k sends the $response_k^2$ message after it has received the corresponding query ($query_j^2$).
- It follows from the *happened before* relation and the previous items that the event $e1$ *precedes* the event $e4$, from which we conclude that p_k was alive when p_i's local clock was equal to τ_i^1, which proves the theorem. □

3.3 An Improvement of the Local Clock-Based Protocol

In the protocol presented in the previous section, when at line 02, a process has received responses from $|est_i.set| - \beta$ processes, it stops waiting and computes the new value of $est_i.set$.

Let us consider the case where $|est_i.set| - \beta$ is small when p_i stops waiting. This can be due to the fact that processes have crashed. This can also be due to the fact that messages from alive processes had not yet arrived, while $\alpha()$ was providing greater and greater values as time was elapsing (line 03). A simple way to correct the bad behavior of the second case is as follows. When it has received responses from $|est_i.set| - \beta$ processes and $|est_i.set| - \beta$ is small, p_i waits during an additional period of time (this period of time can be longer and longer as $|est_i.set| - \beta$ becomes smaller and smaller) in the hope to receive more responses. If it receives additional responses, it can consider them when it computes $est_i.set$. Otherwise, its behavior is unchanged. (It is easy to see that as the system is asynchronous, the proofs of the theorems remain valid when the protocol includes such an additional finite waiting period.)

Remark: When n is Unknown. Let us consider the case of the asynchronous systems where the processes are provided with a broadcast operation that allows each process to send the same message to the rest of processes in the system (e.g., like Ethernet networks, radio networks, or IP-multicast). Interestingly, in such a context, none of the previous protocols requires the specific knowledge of the number n of processes (the size n arrays of the second protocol can be replaced by unbounded arrays or lists). This means that the code of both protocols is independent of the system size.

4 Simulation-Based Experimental Evaluation

As indicated in Section 1, in a pure asynchronous system, there is no way for a process to know whether a given process has crashed or is only very slow. The computation model considered in the previous sections allows only each process to: 1) use a local clock to measure time durations and 2) invoke a predefined $\alpha()$ function. It is not powerful enough to allow circumvention of the previous impossibility result.[3] As we have seen, the previous protocols "only" provide each process with a *local estimate* of the processes that are alive. The formal properties associated with these estimates are stated in Theorems 1-4.

So, an important issue concerns the practical relevance of the estimate sets that are computed. This can be summarized in the following question: "how *accurate* is the estimate each process is provided with by the protocol?" To answer this question, simulation-based experiments have been realized. The simulation considers the protocol based on nonsynchronized local clocks (Fig. 2).

4.1 The Simulation Model

The answer to the previous question depends on several parameters including the communication subsystem, the distribution of the message transmission delays, the crash pattern, and the definition of the $\alpha()$ function.

The Underlying Network. The simulator executes a sequence of rounds. A round corresponds to an execution of the operation estimate() by each process. At each round, the simulator computes randomly (using a normal distribution law) the transfer delays of the messages. At the end of a round, each process obtains a local estimate of the set of the processes that are alive. This estimate is then used by the protocol to bound the number of responses that the process waits for during the next round.

In order to simulate a realistic communication subsystem, the simulation is as follows. The network is defined by routers. The processes and the routers are randomly distributed in a two-dimensional geometric space (uniform distribution law). There are a given number of routers and each process is connected to its closest router (so, no two processes communicate directly, some processes communicate through one router, while other processes communicate through several routers). The communication between routers is assumed to be more costly than the communication between a process and its router. The simulator considers three routers and assumes that the communication between two routers is three times more costly than the communication between a process and the router it is connected to. (In fact, we run our simulator with different numbers of routers, and the conclusion was that the number of routers—despite the fact that this number is much smaller than the number of processes— does not affect the results in a noteworthy manner.)[4]

Crash Pattern and Function $\alpha()$. The simulation considers $n = 100$ processes, and the worst-case scenario for the crash pattern. As we are interested in measuring the false suspicions, the worst-case scenario is when no process crashes. This is because a process p_i wrongly suspects a process p_j (to have crashed) as soon as p_j does not appear in the estimate set of p_i.

3. Allowing a process to safely know which processes are crashed and which process are alive requires a perfect failure detector [3], [17], and the implementation of such a failure detector requires a stronger additional equipment than local clocks and $\alpha()$.

4. There is, however, an exception for the case of only two routers. In that case, due to communication delays between the routers, the network can momentarily behave as a partitioned network.

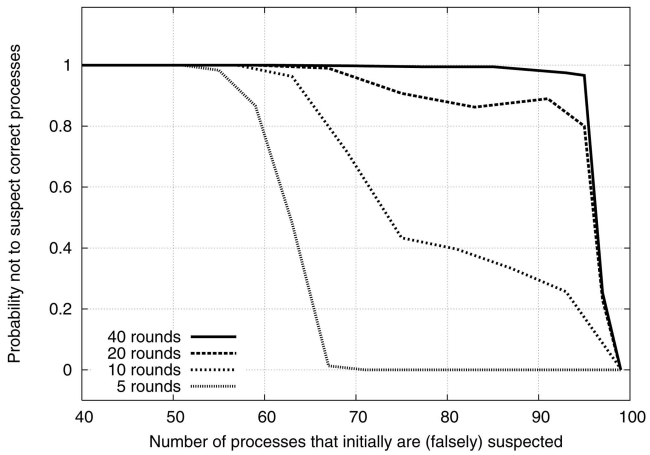

Fig. 4. Probability not to suspect correct processes.

The function $\alpha()$ used in the simulation considers that one process can crash per time unit. It is important to notice that, as a time unit is defined with respect to communication, this definition of $\alpha()$ favors erroneous suspicions.

4.2 How Accurate is the Protocol?

Accuracy of an Estimate. We have seen (Theorem 1) that a crashed process is eventually suspected. So, from a user point of view, we define the *accuracy* of the protocol as the probability that each estimate set it computes contains the processes that are currently alive. (If the application layer considers the assumption "no estimate set misses correct processes," the accuracy notion can be seen as a measure of the *coverage* of that assumption [15].)

The following experiments have been realized to determine the accuracy provided by the protocol. In order to measure the probability not to suspect a correct process after the execution of x rounds, each run of the simulation considers that initially, each process wrongly suspects some number of correct processes. The results are depicted in Fig. 4.

What We Learn from Fig. 4. Fig. 4 represents four curves for different values of x namely, 40, 20, 10, and 5 rounds. (Let us notice that the number of rounds can be interpreted as the duration needed by the protocol to provide each process with an accurate estimate set.)

Fig. 4 shows that when, initially, the number of wrong suspicions does not bypass the majority of processes, the probability for a process to have an estimate including all processes after only five rounds is practically equal to one. This can be seen as the *normal* case.

In *unstable* periods (i.e., when the communication delays are particularly erratic), an estimate set can miss correct processes. This situation can be seen as if the protocol starts with initial wrong suspicions. The figure shows that when such a period terminates, the estimate of a process does not miss correct processes after $x = 40$ rounds even if initially, each process falsely suspects 80 processes. (The figure also shows that this can be obtained in much less rounds when there are less initial wrong suspicions, which is the case, in practice).

Interestingly, Fig. 4 also shows that the only cases where the protocol is unable to compute accurate estimate sets in a reasonable number of rounds is when the number of initial wrong suspicions is around 95 percent of the processes (i.e., in a case that practically never occurs).

4.3 How Fast is the Protocol Convergence?

Another important question concerns the convergence speed (measured in number of rounds) of the protocol. This issue is addressed in Figs. 5a and 5b.

What We Learn from Fig. 5a. Assuming an initial number of wrong suspicions (horizontal axis), Fig. 5a gives the number of rounds (vertical axis) required for a process to obtain an estimate that does not miss correct processes. As an example, the figure shows that if a process initially misses up to 55 percent of the correct processes, these wrong suspicions are corrected only after two rounds. Then the number of rounds naturally increases. If a process initially misses up to 80 percent of the correct processes, these wrong suspicions are corrected only after 13 rounds. The number of rounds drastically increases only when the number of initial suspicions bypasses 95 percent of the processes.

What We Learn from Fig. 5b. Fig. 5b offers another view of the properties of the protocol. The simulation that produced the results depicted in this figure considers that the number of processes in the system equal to 50. Each curve considers a particular value for the number of

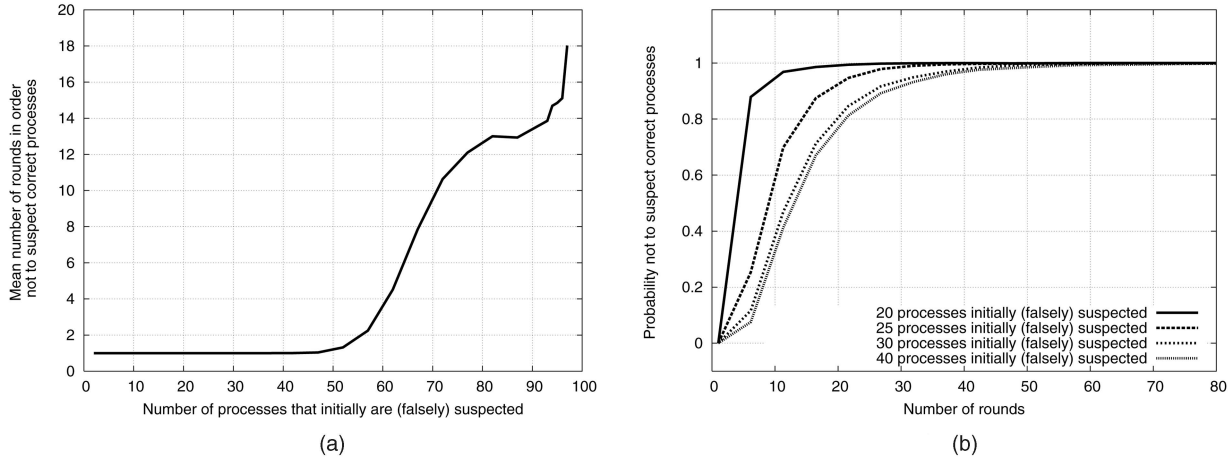

Fig. 5. Simulation results: How fast is the protocol to provide an accurate estimate.

processes that are initially wrongly suspected, namely, 40, 30, 25, and 20. It gives the probability to have an accurate estimate (no false suspicions) according to the number of rounds that are performed. It is interesting to notice that the four curves converge to one independently from the number of initial wrong suspicion. Moreover, the shape of the curves shows that this convergence is obtained quickly. Let us also observe that, when the number of initial wrong suspicions is less than the majority (which does correspond to both the higher curves), the convergence is very fast. So, this figure summarizes the two previous ones. It shows that: 1) after some number of rounds an estimate contains all the processes that are alive (accuracy) and 2) this number of round is relatively small (fast convergence).

5 A System Model with Both t and $\alpha()$

Let us consider the traditional asynchronous message-passing model where the model parameter t is an upper bound on the number of processes that might crash (which means here that "by assumption," there is no execution in which more than t processes crash). The previous results encourage us to enhance this model by considering a model providing both the parameter t and the function $\alpha()$.

Let us consider the following basic statement usually used after the broadcast of a query in the traditional asynchronous message-passing model:

"**wait until** $(n-t)$ messages have been received."

As noticed in the introduction, this is the "best" that can be done with respect to the response quality criterion when the only "additional knowledge" on the system behavior is the parameter t. In the proposed extended model, this basic statement can be left unchanged, or it can be rewritten as follows:

"**wait until** $\max(|est_i.set| - \alpha(\Delta), n-t)$ messages have been received,"

where $est_i.set$ and Δ are provided by the underlying protocol described in Section 3.

It is important to notice that when we consider the number of messages received as the quality of service criterion associated with the **wait until** statement, the behavior of the extended system model is usually better, and never worse, than the one provided by the classical t-based system model. This follows directly from the fact that for each **wait until** statement, the number of messages received: 1) is never less than $n-t$ and 2) is usually greater according to the current value of $|est_i.set| - \alpha(\Delta)$. In that sense, the extended system model provides a better quality of service at a "small" additional price (namely, including the function $\alpha()$). Moreover, both waiting conditions (with or without $\alpha()$) can be used by an upper layer application according to its needs.

6 Wait Longer to Improve Accuracy

This section studies the impact of the structure on the underlying network. It presents a simple but practically efficient technique to cope with this impact.

6.1 Impact of the Underlying Network

In the experiments described in Section 4, the processes communicate through a network composed of interconnected routers. This underlying communication structure is such that, when two processes p_i and p_j are connected to different routers, the messages they exchange suffer a router-to-router delay. To take into account these routing latencies, the simulator used in Section 4 adds a delay to each message that uses more than one router to arrive at its destination process [20]. Considering such a context, Fig. 5a has presented the average number of rounds needed by the protocol to ensure that no correct process falsely suspects another correct process, this number of rounds being defined as a function of the number of false suspicions initially introduced. The curve presented in this figure has been obtained for a given setting (namely, three routers, and a constant router-to-router delay).

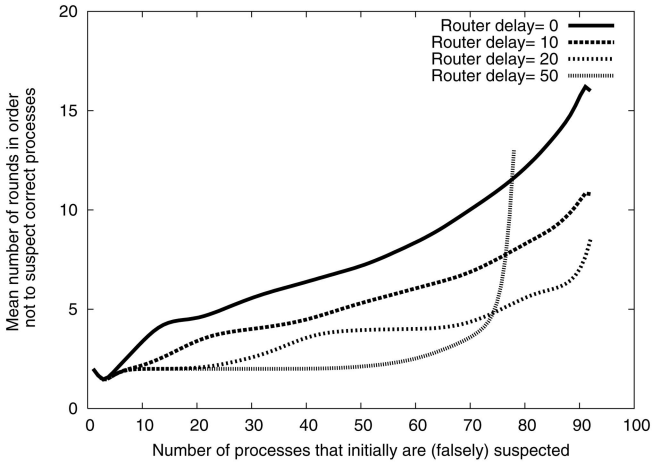

Fig. 6. Convergence time for various router-to-router delays.

The present section investigates the impact of the router-to-router delay on the convergence time (always expressed as a number of rounds). Fig. 6 presents different curves, obtained from executions of the protocol described in Section 3.2. Each curve is associated with a particular router-to-router delay. When looking at the four curves, we see that the higher the router-to-router delay, the lower the convergence time. However, when the router-to-router delay, exceeds some threshold, the convergence time increases exponentially (lowest curve in the figure) when the initial number of false suspicions exceeds 70 percent.

These observations can be explained by the amount of new information a process obtains at each round. With a small router-to-router delay, all the processes are virtually connected to the "same router" and they all belong to the same cluster. Consequently, processes are discovered in a (nearly) random fashion as each process waits for some number of messages and different messages have the same probability to be included in its set. Conversely, with a long delay, a process receives messages first from processes connected to the router it is connected to (i.e., from processes that are in the same cluster), and only then from processes connected to other routers (i.e., from other clusters).

The curves depicted in Fig. 6 represent two extreme cases that may impact performances. In the simulations, the delay from a process to a router is (on average) 35 time units, and a router-to-router delay costs from 0 to 50 time units for the different curves (respectively, 0, 10, 20, and 50 time units for the four curves, as indicated at the top left of the figure). The optimal configuration lies between the two extreme cases depicted in the figure (highest and lowest curves). As shown in the figure, a system whose router-to-router delay is 20 time

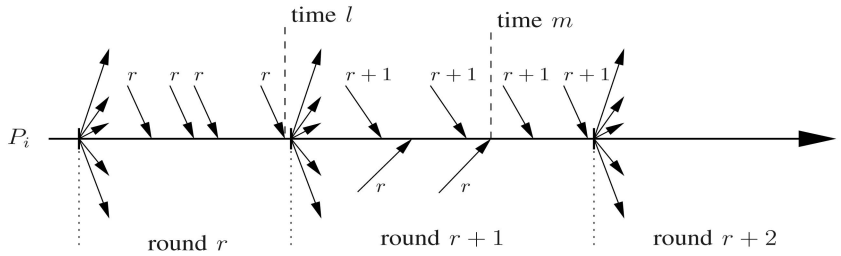

Fig. 7. Arrival time of messages.

units converges quickly (and nearly always). Unfortunately, this delay is a parameter that is not under the control of the system administrator.

6.2 Improving the Protocol

As just seen, the processes that are within the same cluster (i.e., connected to the same router or close to each other in terms of number of routers) have shorter communication delays than processes connected by more routers. As shown in Fig. 6, this difference in communication delays may entail a very long convergence time or even prevent the proposed protocol from converging in "bad" settings (i.e., when initially there is a high number of false suspicions). This section shows that it is possible to remedy this problem in a relatively simple way.

Let us first recall that the processes have no a priori information on the structure of the communication network and on router-to-router transfer delays. The idea is to force every process to wait during some period of time (determined locally as shown later) before sending messages at the beginning of each round. Of course, if all the processes delay their messages in the same way, the status quo will be maintained. The simple remedy to that problem consists in favoring a small number of messages exchanged by distant processes by establishing "small world"-like connections.

To that end, two local dates are associated with message arrivals at each process. Let us notice that a message sent at round $r$can be received by p_i, while that process is executing any round r'. These local dates are defined as follows.

- ℓ_r^r is the arrival date (measured by p_i's local clock) of the last round r message received by p_i, while it is executing its round r (i.e., still waiting for round r messages, as depicted in Fig. 7).
- It is possible that messages sent during round r arrive at p_i, while that process has already started its round $r+1$([5]). ℓ_r^{r+1} is the local arrival date of the last message (sent in round r) that is received by p_i, while it executes round $r+1$ (see Fig. 7). If no round r message is received by p_i, while it executes its round $r+1$, we have $\ell_r^{r+1} = \ell_r^r$.

The pair of values (ℓ_r^r, ℓ_r^{r+1}) computed by a process p_i can be used by the protocol, executed by p_i, for the round $r+2$. More precisely, process p_i delays the sending of its messages for a time period uniformly drawn from the interval $[0, \ell_r^{r+1} - \ell_r^r]$. The aim of these additional delays is to favor the on time arrival of messages sent by distant processes, and consequently, improve the protocol accuracy despite the heterogeneity of the underlying network configuration.

Fig. 8 illustrates the benefit of using the dates ℓ_r^r and ℓ_r^{r+1}. The simulation results depicted in that figure correspond to the execution of the protocol presented in Section 3.2 modified as explained above. The curves have been obtained in the same setting as in Fig. 6 (that corresponds to the base protocol). Comparing both figures shows that the improved protocol allows the system to converge faster in all situations without requiring any knowledge of the structure of the underlying network. In particular, systems with a huge router-to-router delay (that could entail a partitioned view of the system when there are too many false suspicions) now converge in few rounds even when there is as much as 90 percent of false initial suspicions.

Remark. It is possible to analyze these results from a *small world* point of view [10]. Each router defines a cluster whose processes are tightly connected in the sense that any two processes in the same cluster can be considered as *local neighbors*. Adding a random delay to the waiting period allows processes from different clusters to wait long enough to receive messages from other clusters. This can be seen as adding long-range neighbors (*shortcuts* in the small world terminology [10]), as opposed to local neighbors. Such shortcuts are known to be very useful for routing. The proposed protocol shows that they can be simulated to obtain more efficient exchanges of messages informing which processes are alive in the system.

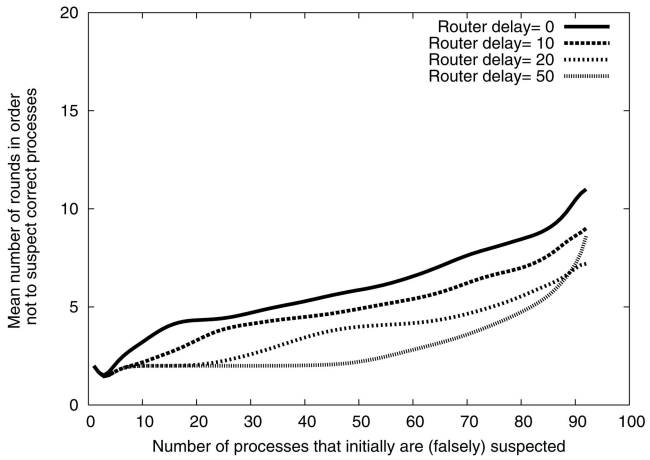

Fig. 8. Convergence time for various router-to-router random delays.

5. In that case, it is possible that; when p_i has computed the waiting period of its round $r+1$, it had only a partial view of the set of processes that were alive at round r.

7 Conclusion

This paper focused on on the fly determination of which processes are alive in an asynchronous-distributed system. To that end, the paper proposes to replace the traditional parameter t (that is assumed to define an upper bound on the number of processes that can crash during an execution) by a function (denoted as $\alpha(\Delta)$) that returns an estimate of the number of processes that can crash during a period of Δ time units. The paper has proposed two protocols. The first is based on a global clock. The second uses only nonsynchronized local clocks (a local clock is used only to allow a process to measure durations). A simulation study has shown that these protocols ensure a pretty good quality of service in the sense that an alive process never remains suspected for a long time, while the crashed processes are quickly suspected. Finally, the paper has also addressed the case of heterogeneous communication networks.

References

[1] H. Attiya and J. Welch, *Distributed Computing: Fundamentals, Simulations and Advanced Topics,* second ed. Wiley-Interscience, p. 414, 2004.

[2] M. Ben-Or, "Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols," *Proc. Second ACM Symp. Principles of Distributed Computing (PODC '83),* pp. 27-30, 1983.

[3] T. Chandra and S. Toueg, "Unreliable Failure Detectors for Reliable Distributed Systems," *J. ACM,* vol. 43, no. 2, pp. 225-267, 1996.

[4] F. Cristian and C. Fetzer, "The Timed Asynchronous Distributed System Model," *IEEE Trans. Parallel Distributed Systems,* vol. 10, no. 6, pp. 642-657, June 1999.

[5] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the Presence of Partial Synchrony," *J. ACM,* vol. 35, no. 2, pp. 288-323, 1988.

[6] C. Fetzer, "Perfect Failure Detection in Timed Asynchronous Systems," *IEEE Trans. Computers,* vol. 52, no. 2, pp. 99-112, Feb. 2003.

[7] C. Fetzer, M. Raynal, and F. Tronel, "An Adaptive Failure Detection Protocol," *Proc. Eighth IEEE Pacific Rim Int'l Symp. Dependable Computing (PRDC '01),* pp. 146-153, 2001.

[8] M.J. Fischer, N.A. Lynch, and M.S. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *J. ACM,* vol. 33, no. 2, pp. 374-382, 1985.

[9] N. Hayashibara, X. Defago, R. Yared, and T. Kayatama, "The ϕ Accrual Failure Detector," *Proc. 23rd Int'l IEEE Symp. Reliable Distributed Systems (SRDS '04),* pp. 66-78, 2004.

[10] J. Kleinberg, "The Small-World Phenomenon: An Algorithmic Perspective," *Proc. 32nd ACM Symp. Theory of Computing (STOC '00),* pp. 163-170, 2000.

[11] S. Krishnamurthy, W.H. Sanders, and M. Cukier, "An Adaptive Quality of Service Aware Middleware for Replicated Services," *IEEE Trans. Parallel and Distributed Systems,* vol. 14, no. 11, pp. 1112-1125, Nov. 2003.

[12] L. Lamport, "Time, Clocks and the Ordering of Events in a Distributed System," *Comm. ACM,* vol. 21, no. 7, pp. 558-565, 1978.

[13] N.A. Lynch, *Distributed Algorithms,* p. 872. Morgan Kaufmann Publishers, 1996.

[14] A. Mostefaoui, E. Mourgaya, and M. Raynal, "Asynchronous Implementation of Failure Detectors," *Proc. Int'l IEEE Conf. Dependable Systems and Networks (DSN '03),* pp. 351-360, 2003.

[15] D. Powell, "Failure Mode Assumptions and Assumption Coverage," *Proc. 22nd Int'l IEEE Symp. Fault-Tolerant Computing (FTCS-22),* pp. 386-395, 1992.

[16] M. Rabin, "Randomized Byzantine Generals," *Proc. 24th IEEE Symp. Foundations of Computer Science (FOCS '83),* pp. 116-124, 1983.

[17] M. Raynal, "A Short Introduction to Failure Detectors for Asynchronous Distributed Systems," *ACM SIGACT News, Distributed Computing Column,* vol. 36, no. 1, pp. 53-70, 2005.

[18] M. Raynal and F. Tronel, "Group Membership Failure Detection: A Simple Protocol and Its Probabilistic Analysis," *Distributed Systems Eng.,* vol. 6, no. 3, pp. 95-102, 1999.

[19] R. Van Renesse, Y. Minsky, and M. Hayden, "A Gossip-Style Failure Detection Service," *Proc. IFIP Int'l Conf. Distributed Systems Platforms and Open Distributed Processing (Middleware),* 1998.

[20] H. Zhang, A. Goel, and R. Govindan, "An Empirical Evaluation of Internet Latency Expansion," *ACM SIGCOMM Computer Comm. Rev.,* vol. 35, no. 1, pp. 93-97, 2005.

Achour Mostefaoui received the Engineer degree in computer science from the University of Algiers in 1990, and the PhD in computer science from the University of Rennes, France, in 1994. He is currently an associate professor at the Computer Science Department of the University of Rennes, France. His research interests include fault-tolerant distributed systems, group communication, consistency in DSM systems, and distributed checkpointing. He has published more than 100 scientific publications and served as a reviewer for more than 30 journals and international conferences. He is heading the Master of Computer Science on Security of Information Systems of the University of Rennes.

Michel Raynal received the Doctorat dEtat in computer science from the University of Rennes, France, in 1981. He is currently a professor of computer science in IRISA (CNRS-INRIA-University Joint Computing Research Laboratory located in Rennes), where he founded a research group on distributed algorithms in 1983. His research interests include distributed algorithms, distributed computing systems, and dependability. His main interest lies in the fundamental principles that underlie the design and the construction of distributed computing systems. He has been the principal investigator of a number of research grants in these areas, and has been invited by many universities all over the world to give lectures and tutorials on distributed algorithms and fault-tolerant distributed computing systems. He belongs to the editorial board of several international journals. He has published more than 100 papers in journals and more than 210 papers in international conferences. He has also written seven books devoted to parallelism, distributed algorithms, and systems (MIT Press and Wiley). He has served in program committees for more than 100 international conferences (including PODC, DISC, ICDCS, DSN, SRDS, etc.) and chaired the program committee of more than 15 international conferences. In 2002-2004, he chaired the steering committee leading the DISC symposium series. He got the IEEE ICDCS Best Paper Award three times in a row: 1999, 2000, and 2001. Recently, he cochaired the SIROCCO 2005 conference and the IWDC 2005 conference. He was the general cochair of ICDCS 2006. He will be the program chair of OPODIS 2009 and the conference cochair of ICDCN 2010.

Gilles Tredan received the Engineer and MS degrees in computer science in 2006 from the University of Rennes, France, where he is currently working toward the PhD degree under the supervision of Achour Mostefaoui at the Computer Science Department. His research interests include distributed algorithms for sensor and peer-to-peer networks, security and byzantine behaviours, and distributed connectivity and centrality monitoring.