On the fly estimation of the processes that are alive/crashed in an asynchronous message-passing system

Achour MOSTEFAOUI Michel RAYNAL Gilles TREDAN
IRISA, Université de Rennes 1, Campus de Beaulieu, 35042 Rennes Cedex, France
{achour|raynal|gtredan}@irisa.fr

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. To that end, it considers a realistic computation model where the processes are provided with non-synchronized local clocks and a function $\alpha()$. That function takes a local duration as a parameter, and returns an integer that is an estimate of the number of processes that can crash during that duration. A simulation-based experimental evaluation of the protocol is also presented. The experiments show that the protocol is practically relevant.

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 an algorithm that allows a process to determine an estimate of the number of crashes. An easy algorithm 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). Moreover, let a process answers an inquiry message by 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 under-estimate as a process can crash just after having sent its answer.

Differently, 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) [5]. 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 timeout values it uses. The impossibility to distinguish a crashed process from a process with which communication are very slow is one of the main difficulties one has to face when interested in designing distributed fault-tolerant services. The most celebrated related result is the impossibility to solve the consensus problem in asynchronous systems prone to even a single process crash failure [3].

The classical asynchronous distributed system model Let n be the number of processes the system is made up of (we consider here only 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 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 comprised between $n - t$ and $n - f$.
- If $f > t$, the querying process blocks forever. This 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 it 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 main challenges one has to take up, as soon as we want to ensure a good response quality while preventing process permanent blocking.

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 [3]. Ways to circumvent this impossibility are presented in [1] (failure detector approach; see [10] for a survey of this approach), [2] (additional synchrony), and [9] (randomized algorithms).

Content of the paper The paper is on the on the fly determination 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 can crash during a given time duration (taking always t would provide the classical asynchronous model).
  More precisely, the model assumes that the processes are provided with non-synchronized local clocks and can invoke a function (denoted $\alpha()$) 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 algorithm is presented and proved correct. That algorithm provides each process with an estimate of the processes that are currently alive. The proposed algorithm is based on a simple query-response mechanism and the local clock of the querying process (to make easier the presentation, a global clock-based algorithm is first presented). That algorithm 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.

Roadmap The paper is made up of 5 sections. Section 2 presents the computation model. Then Section 3 introduces two distributed algorithms that provide each process p with a set containing the processes that p can consider as the processes currently alive. The presentation of these algorithms is incremental: the first algorithm 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 non-synchronized local clocks. Then, Section 4 presents a simulation study of the local clock-based algorithm. It shows that the algorithm 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 non-suspected. Section 5 proposes an extended computation model including both the parameter t and $\alpha()$.

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 an 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 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 non-trivial 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 designing correct and non-blocking protocols (as we have seen in the introduction, 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 $\alpha()$ 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 can crash during Δ time units. As we can see, 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:

- $\Delta_1 \geq \Delta_2 \Rightarrow \alpha(\Delta_1) \geq \alpha(\Delta_2)$ (non-decreasing).
- 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 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/crashed processes

This section presents an algorithm 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 understanding easier, the algorithm is presented in two steps. We first assume that the processes have access to a common global clock. Then, that global clock is approximated with the non-synchronized 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 algorithm

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 Figure 1, that 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 $est_i.date$. $est_i.set$ 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 $rec\_from_i.set$ where 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.

[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 other processes (varying from 0% up to 95%).

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 [6]: p_i issues a query (line 02) and waits for corresponding responses (line 03). 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 correspond 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 03-04). This is where the $\alpha()$ function provided by the model comes into play. Until it stops waiting, p_i repeatedly evaluates $\beta = \alpha(\text{global\_clock}() - est_i.date)$ (line 04). The current value of β is an approximation of the the number of processes that could 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 crash since that time (line 03). (As already indicated, the responses that arrive too late are discarded.)

Properties It is important tot 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.

The following theorems state the properties provided by the algorithm. Theorems 1 and 2 define the safety property ensured by the algorithm. 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 algorithm.

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 $est_i.set$. $\square_{Theorem\ 1}$

Theorem 2 *Let us consider an invocation* estimate() *issued by a process p_i. None of the processes in the $est_i.set$ returned by 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 $est_i.date$ at line 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. $\square_{Theorem\ 2}$

Theorem 3 *Let us consider an execution in which p_i is a correct process. p_i never blocks forever in the* **wait until** *statement (line 03-04).*

Proof Let us assume that a correct process p_i blocks forever in the **wait until** 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 $|est_i.set| - \beta \leq 1$ 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. $\square_{Theorem\ 3}$

```
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
```

Figure 1. The estimate() operation (version based on global time)

3.2 A local clock-based algorithm

This section adapts the previous algorithm to a setting without global clock, each process being provided only with a local clock. As the local clocks are not synchronized, 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 03-04 of the global time-based algorithm described in Figure 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 previously. 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 algorithm.
- 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 Figure 2). When, it receives that time value (that is meaningful only for p_j) p_i stores it $last\_date_i[j]$ (line 05). In that way, p_i is able to indicate to p_j the date (measured with p_j'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 Figure 2. It is nearly the same as the behavior defined for the global time-based algorithm. 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 computes the date $est_i.date$ it associates with the set $est_i.set$ (line 07).

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

Theorem 4 *Let us consider an invocation* estimate() *issued by a process p_i as described in Figure 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 Figure 3 where two queries issued by p_j are described such that the

```
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
```

Figure 2. The estimate() operation (version based on non-synchronized local time)

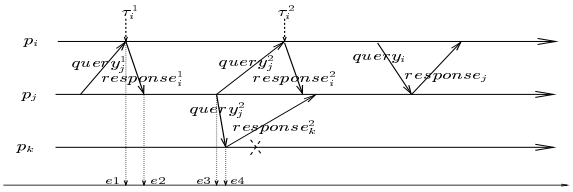

Figure 3. Using the *happened before* relation

two corresponding response messages sent by p_i are processed by p_j. Let τ_i^1 and τ_i^2 the current value of p_i's local clock when that process sent the corresponding response messages.

Due to the algorithm, 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 05-06 executed by p_j).
- $p_k \in rec\_from_j$ (line 04 executed by p_j).

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 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 algorithm, 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 local clock was equal to τ_i^1. This follows from the following observation based on Lamport's *happened before* relation [4].

- 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 local clock was equal to τ_i^1, which proves the theorem.

$\square_{Theorem\ 4}$

Improvements of the algorithm are described in [7].

4 Simulation-based experimental evaluation

As indicated in the introduction, in a pure asynchronous system, there is no way for a process to know whether a given process has crashed or 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[2]. It is not powerful enough to allow circumventing the previous impossibility[3]. As we have seen, the previous algorithms "only" provide each process with a *local estimate* of the processes that are alive. The formal properties associated with these estimates are stated by the 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 algorithm?" To answer this question simulation-based experiments have

[2] Let us notice that the outputs of the $\alpha()$ could be arbitrary, e.g., always equal to $n-1$. (but such "bad" values would have an impact on the "quality" of the estimate sets that are computed).

[3] Allowing a process to safely know which processes are crashed and which process are alive require a perfect failure detector [1, 10], and the implementation of such a failure detector requires a stronger additional equipment than local clocks and $\alpha()$.

been realized. The simulation considers the algorithm based on non-synchronized local clocks (Figure 2).

4.1 The simulation model

The answer to the previous question depends on several parameters including the communication sub-system, 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 algorithm 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.

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.

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 algorithm

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 algorithm 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 [8].)

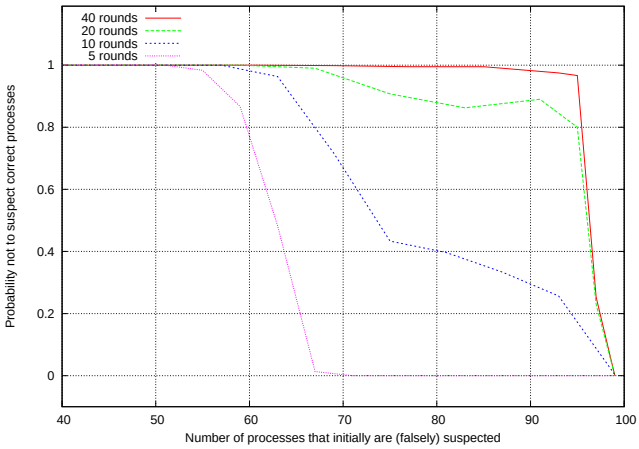

Figure 4. Proba to correct false suspicions

The following experiments have been realized to determine the accuracy provided by the algorithm. 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 Figure 4.

What we learn from Figure 4 Figure 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 algorithm to provide each process with an accurate estimate set.)

Figure 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 5 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 algorithm 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, Figure 4 also shows that the only cases

where the algorithm is unable to compute accurate estimate sets in a reasonable number of rounds is when the number of initial wrong suspicions is around 95% of the processes (i.e., in a case that practically never occurs).

4.3 How fast is the algorithm convergence

Another important question concerns the convergence speed (measured in number of rounds) of the algorithm. This issue is addressed in Figure 5.

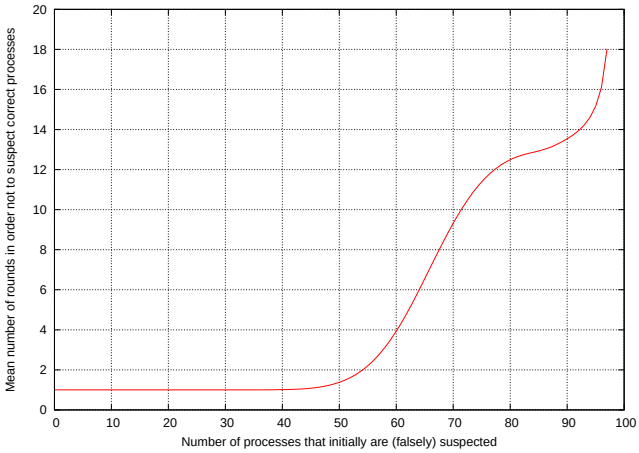

Figure 5. Measuring how fast is the algorithm to provide an accurate estimate

What we learn from Figure 5 Assuming an initial number of wrong suspicions (horizontal axis), Figure 5 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% of the correct processes, these wrong suspicions are corrected after only two rounds. Then the number of rounds naturally increases. If a process initially misses up to 80% of the correct processes, these wrong suspicions are corrected after only thirteen rounds. The number of rounds drastically increases only when the number of initial suspicions bypasses 95% of the processes.

5 A system model including 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 there is no system 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 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 algorithm 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.

References

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

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

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

[4] Lamport L., Time, Clocks and the Ordering of Events in a Distributed System. *Comm. ACM*, 21(7):558-565, 1978.

[5] Lynch N.A., Distributed Algorithms. *Morgan Kaufmann Pub.*, San Francisco (CA), 872 pages, 1996.

[6] Mostefaoui A., Mourgaya E., and Raynal M., Asynchronous Implementation of Failure Detectors. *Proc. Int'l IEEE Conference on Dependable Systems and Networks (DSN'03)*, IEEE Computer Society Press, pp. 351-360, 2003.

[7] Mostefaoui A., Raynal M. and Tredan G., On the fly estimation of the processes that are alive/crashed in an asynchronous message-passing system. *Tech Report*, IRISA, University de Rennes, France, 2006.

[8] Powell D., Failure Mode Assumptions and Assumption Coverage. *22nd Int'l IEEE Symposium on Fault-Tolerant Computing (FTCS-22)*, IEEE Press, pp.386-395, 1992.

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

[10] Raynal M., A Short Introduction to Failure Detectors for Asynchronous Distributed Systems. *ACM SIGACT News, Distributed Computing Column*, 36(1):53-70, 2005.