Impact of a Failure Detection Mechanism on the Performance of Consensus

Nicole Sergent[†,*] nicole.sergent@axs-tech.com
Xavier Défago[‡,*] defago@jaist.ac.jp
André Schiper[*] andre.schiper@epfl.ch

[†] *AXS Technologies, 1003 Lausanne, Switzerland*
[‡] *Japan Advanced Institute of Science and Technology (JAIST), 1-1 Asahidai, Tatsunokuchi, Ishikawa 923-1292, Japan*
[*] *École Polytechnique Fédérale de Lausanne (EPFL), 1015 Lausanne, Switzerland*

Abstract

The paper considers a consensus algorithm for an asynchronous system augmented with failure detectors, and analyze the impact on its termination time of various implementations of failure detectors. This study shows that the design of fault-tolerant distributed algorithms in the asynchronous system model augmented with failure detectors is orthogonal *to implementing the actual failure detectors. This nicely decouples logical issues (proof of correctness) from engineering issues (e.g., performance and timing constraints).*

1. Introduction

Fault tolerance is a very important concern in distributed systems. It is usually achieved by introducing a certain degree of redundancy, either in time or in space. A common approach consists in replicating the vulnerable components of a system. Although an intuitive concept, replication poses difficult problems, requires sophisticated techniques, and has thus generated a large amount of literature.

In distributed systems, many problems require the participating processes to coordinate their decisions. More specifically, the problem of consensus in the presence of failures is central in the context of fault tolerant distributed systems. However, in purely asynchronous systems, the problem of consensus is not solvable in the presence of failures [8]. The intuition for this impossibility is related to the fact that, in an asynchronous system, it is impossible to distinguish a crashed process from a very slow one.

In order to solve consensus, Chandra and Toueg [2] have extended the asynchronous system model with the notion of failure detectors. In their work, they identify families of failure detectors that are defined according to their respective properties. With Hadzilacos [1], they have shown that the failure detector $\Diamond\mathcal{S}$ is the weakest failure detector that allows us to solve consensus in asynchronous systems. The work of Chandra and Toueg shows that, given a failure detector with adequate properties, the consensus is solvable in an asynchronous system. Interestingly, their consensus algorithm for $\Diamond\mathcal{S}$ requires a failure detector for *liveness*, but it always ensures *safety*. In other words, if the algorithm is run in a purely asynchronous system, then, even though it is not guaranteed to terminate, it will never violate any of the safety properties of consensus.

The goal of the paper is to show that, although the failure detector model[1] is essentially a time-free model, solving consensus in this model still allows us to consider timing issues. In the paper, we consider the consensus algorithm based on the failure detector $\Diamond\mathcal{S}$ [2], and various implementations of this failure detector. Our goal is then to find the failure detector implementation that leads to the fastest solution of con-

[*] Sergent and Défago were affiliated with the EPFL when the present research work was initiated. This research is currently supported both by EPFL and JAIST.

[1] From here on, the “failure detector model” means the “asynchronous system model augmented with failure detectors”.

sensus in the two more frequent cases: (1) failure free execution, and (2) worst case of a single process crash.

As a result, the paper shows that the failure detector model does not prevent from analyzing timing issues. On the one hand, the model allows for a clean separation of “logical issues” (proof of safety and liveness of the algorithm) from “engineering issues” (implementation of failure detectors). On the other hand, when timing issues have to be considered, the implementation of the failure detectors is considered in combination with the algorithm.

The rest of the paper is structured as follows. Section 2 gives a short overview of the background of this work. Section 3 describes the “contention aware” model that we use to evaluate the performance of distributed algorithms using simulation. Section 4 describes different implementations of the failure detectors. Section 5 analyses the influence of the implementation of failure detectors on the termination time of a consensus algorithm. Finally, Section 6 discusses related works, and Section 7 concludes the paper.

2. Background

2.1. Consensus

Consensus is a central problem in the context of distributed systems, being at the heart of many agreement problems [2, 10, 9]. The problem is defined on a set Π of processes: every process $p_i \in \Pi$ starts with an initial value v_i, and all correct processes must agree on a common value v that is the initial value of one of the processes.

2.2. Failure detector model

The failure detector model considers a module FD_p attached to every process p, whose role is to give information about the status “alive/crashed” of every other process in the system [2]. Typically, a failure detector maintains a list of processes that it suspects to have crashed. A failure detector can make mistakes by incorrectly suspecting a correct process. Nevertheless, whenever a failure detector discovers that some process was incorrectly suspected, it can change its mind.

The failure detectors are grouped into different classes, according to their properties. It has been

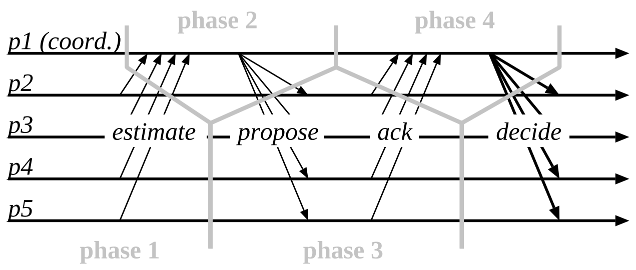


Figure 1. Consensus algorithm when no failure or suspicion occur ($n = 5$ processes).

shown that $\Diamond\mathcal{S}$ is the weakest failure detector class allowing us to solve the consensus problem [1]. This result implies that $\Diamond\mathcal{S}$ captures the minimal amount of synchrony needed to solve consensus in the presence of failures. This result also applies to problems that can be solved by a transformation to a consensus problem, e.g., atomic broadcast [2].

2.3. Solving consensus with $\Diamond\mathcal{S}$

In this paper, we consider the consensus algorithm using the failure detector $\Diamond\mathcal{S}$ [2]. Understanding the details of this algorithm is not necessary here. However, to give a rough idea, the algorithm proceeds in a sequence of rounds, and in each round another process plays the role of the coordinator of that round. Figure 1 depicts the communication schema of the algorithm, in a failure-free run during which no incorrect suspicion occur. The figure shows the four phases that constitute one round. Informally, the algorithm works as follows.

- In Phase 1, the processes send their estimate to the coordinator.
- In Phase 2, the coordinator waits for a proposition from a majority of the processes and proposes a value chosen among the estimates received.
- In phase 3, the processes wait for a proposition from the coordinator. They adopt the value proposed by the coordinator, acknowledge it (*ack* message), and then proceed to the next round. However, if a process suspects the coordinator before it receives the proposition, it sends a negative acknowledgement (*nack* message) and also proceeds to the next round.

- In Phase 4, the coordinator waits until it has received an acknowledgement (*ack* or *nack* message) from a majority of processes. If the proposition has been acknowledged (*ack*) by a majority of processes, the proposed value becomes the decision value. The coordinator then broadcasts the decision value to the other processes. Otherwise, if the coordinator has received a negative acknowledgement (*nack* message) from one of the processes, it does not send a decision message and directly proceeds to the next round.

A detailed description of this algorithm can be found in [2], together with the proofs. Some practical optimizations to this algorithm are presented in [5].

3. Simulation model

We use discrete event simulation to evaluate different implementations of the failure detectors in the context of a consensus algorithm. There are three main reasons for using simulation rather than actual performance measures. First, in a real system, it is difficult to obtain accurate performance measures for an algorithm that starts and ends on different sites. Second, simulation allows for a fair comparison of the performance of different test cases; the results are obtained using identical assumptions and simulation conditions (this fairness would be impossible to achieve in a real system, where the measurements are biased by unpredictable network and workstation loads). Last but not least, simulation makes it possible to get results faster since it does not need a full-fledged implementation.

3.1. Basic assumptions

Our simulation model is based on the following general assumptions:

- The workstations connected to the network are identical and uniformly distributed along the physical medium.
- The LAN is private, i.e., there are no other message passing on the network beside those generated by the algorithm under study.
- There is only one process running on each workstation.

A process receives messages, sends messages, and does some local computation. We consider that the local computation time is negligible compared to the time needed to transmit messages, i.e., the local computation time is set to 0. Processes communicate via an Ethernet-like network, and use a datagram transport service (UDP/IP). We consider only point-to-point communication.

3.2. Computing message transmission delays

In simulations, message transmission delays are usually computed using some empirical distribution (e.g., [4]). However, such an approach *does not account for the network contention caused by the algorithm itself*. It is therefore not adequate for our study since, in a model that ignores network contention, sending failure detection messages does not slow down the algorithm. In other words, failure detection does not cost anything. Obviously, such a model makes it impossible to evaluate the impact of failure detection mechanisms!

The model that we adopt for point-to-point transmission of messages *takes account of network contention generated by the messages of the algorithm*. This model can be described as follows [13].[2] A message m sent from process p to process q requires the allocation of three resources (Fig. 2):

- the *CPU* resource on the sending host, to execute the UDP/IP send routine;
- the *network*;
- the *CPU* resource on the receiving host, to execute the UDP/IP receive routine.

When one of these resources is needed by a message m and the resource is busy, then m has to wait. CPU resources (for send and receive) are allocated, using a FIFO policy. The network resource is slightly more complex to handle. First, for each host, the messages are handed over to the network according to a FIFO policy. Second, the access to the network is allocated randomly between those hosts that have a message to send.

[2]A similar approach has also been used in [11].

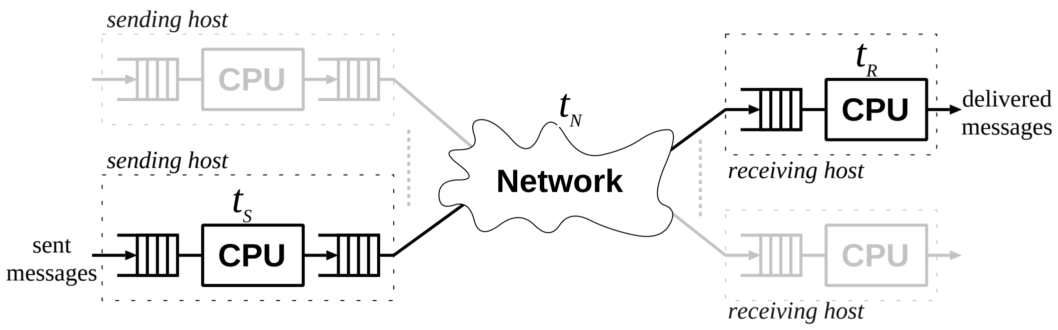


Figure 2. Modeling message transmission

We consider that transmitting a message requires the above three resources for the following constant durations (we assume only messages of a few bytes).

- sending a message requires the CPU of the sending workstation for $t_S = 230\mu$s;
- the network resource is needed for $t_N = 100\mu$s;
- receiving a message requires the CPU of the receiving workstation for $t_R = 250\mu$s.

We have measured these values with Sun SPARCstations-20 connected through 10 Mbit Ethernet, using the technique explained in [13, 12]. We have then validated our message transmission model using the communication schema of a *two-phase commit* protocol (2PC). The simulation results lie within 6% of the experimental results [13],[3] which shows that our model is fairly good.

An important particularity of this model is that it adequately models message contention, both on the network and the CPUs. Based on this model, Urbán *et al* [15] have defined a set of "contention-aware metrics" for distributed algorithms.

4. Failure detection strategies

In this section, we present four different implementations of failure detection. The first two implementations are general techniques, while the last two are algorithm-specific implementations customized for the consensus algorithm. The goal of the two specialized techniques is to reduce the number of "failure detection" messages (which contributes to reduce the termination time of the consensus algorithm).

[3]In order to minimize the the bias from extra load on workstations and network, all measures have been made at unsocial hours.

To simplify the presentation, we do not follow the model of Chandra and Toueg [2], in which the failure detector $\mathrm{FD_p}$ (attached to process p) is distinguished from p: We simply consider that p itself manages the list of processes that it suspects.

4.1. "Heartbeat" implementation

Heartbeat is a well known technique for the implementation of failure detection. Every process q periodically broadcasts a message *"I am alive"* (see Fig. 3). If a process p times-out on some process q, it adds q to its list of suspected processes. If p later receives a message *"I am alive"* from q, then p removes q from its list of suspected processes.

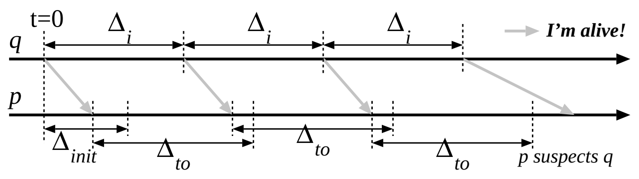


Figure 3. Failure detection: the *heartbeat* implementation.

As illustrated in Figure 3, the failure detector is defined by three parameters: the *initial timeout delay* Δ_{init}, the *heartbeat period* Δ_i and the *timeout delay* Δ_{to}. For the sake of simplicity, we have considered Δ_{init} and Δ_{to} to be equal.

4.2. "Interrogation" implementation

Interrogation is another well known technique for the implementation of failure detection. A process p monitors a process q by sending regularly *"Are you alive?"* messages to q. Upon the reception of such a message, the monitored process replies with an *"I am alive"* message.

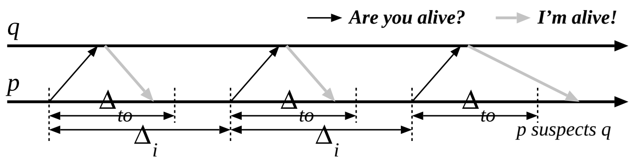


Figure 4. Failure detection: the *interrogation* implementation.

With this implementation, the failure detector is also defined by two parameters: the *interrogation period* Δ_i and the *timeout delay* Δ_{to} (see Fig. 4). Every Δ_i, each process p broadcasts an *interrogation* message (*"Are you alive?"*). Whenever some process q receives an interrogation message from p, it replies with an *"I am alive"* message. If p has not received the *"I am alive"* message from q after a timeout delay Δ_{to}, it adds q to its list of suspected processes. If p later receives a message *"I am alive"* from q, then p removes q from its list of suspected processes.

4.3. Algorithm-specific "silent" implementation

An algorithm-specific implementation of failure detection originates in the observation that failure detection information is needed by some process p only at very specific points during its execution of the consensus algorithm.

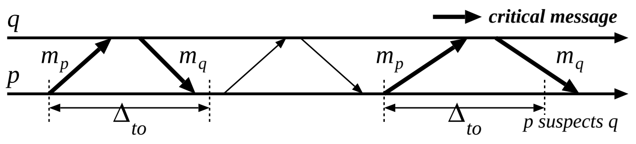


Figure 5. Failure detection: the *algorithm-specific "silent"* implementation.

We can abstract the details of the consensus algorithm by introducing the notion of *critical message* and *critical response* (Fig. 5). The critical message m_p sent by p to q is such that, after having sent m_p, process p waits for message m_q from q. If q crashes, then p must eventually suspect q and stop waiting for m_q. This can be implemented in an ad hoc manner by a simple timeout mechanism, without any additional "failure detection" message. If p does not receive m_q after a delay Δ_{to}, then p suspects q. A similar approach is used by Fetzer and Cristian [7].

This algorithm-specific implementation of failure detection is extremely cheap in terms of messages, as it does not generate any "failure detection" message. In the communication pattern illustrated in Figure 1, the critical message/response pattern occurs when a process, say p_2, sends its estimate to the coordinator p_1 (in Fig. 5; p sends m_p to q), and when p_1 sends back its proposition to p_2 (in Fig. 5; q sends m_q to p).

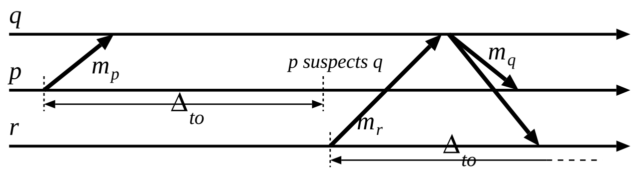


Figure 6. Incorrect suspicions with the algorithm-specific "silent" implementation.

The drawback of this failure detection implementation is the likeliness of incorrect suspicions. Consider Figure 6, where process q waits for message m_p from p *and* message m_r from process r before sending the critical response m_q. If process r is slow, process p may timeout and incorrectly suspect q.

4.4. Application-specific "heartbeat" implementation

The *algorithm-specific heartbeat* implementation overcomes the drawback of the "silent" implementation. The probability of incorrect suspicions is reduced in the following way. Whenever q receives a critical message m_p from a process p, q starts sending *heartbeat* messages to that process. The emission of heartbeat messages stops after q has sent the critical response m_q (Fig. 7).

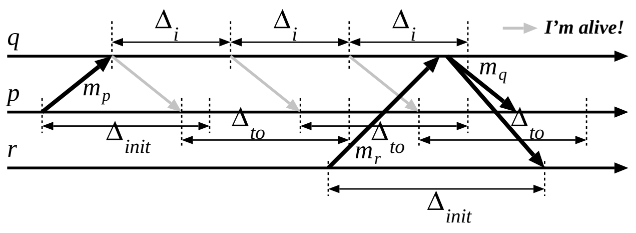


Figure 7. Failure detector: the ***algorithm-specific heartbeat*** **implementation.**

Similar to its generic counterpart, the failure detector is characterized by the three parameters Δ_{init}, Δ_i and Δ_{to}. The parameter Δ_i sets the frequency of the heartbeat messages, while the parameters Δ_{init} and Δ_{to} define the initial and the periodical timeout delays respectively. Again, we have assumed that Δ_{init} is equal to Δ_{to}.

There are certain obvious restrictions on the choice of Δ_i and Δ_{to}. Considering Figure 7, Δ_{to} should be larger than Δ_i or else the heartbeat cannot be received on time. Furthermore, Figure 7 also shows that Δ_{init} should be greater than the average round-trip time.

5. Failure detection strategy and termination time of consensus

In this section, we analyze the impact of the different implementations of failure detection on the termination time of the consensus algorithm presented in Section 2.3, using the simulation model of Section 3. We analyze the termination time in two cases: (1) failure free execution, and (2) worst case (largest termination time) with one process crash.[4] The "failure free" case, and the "one crash" case represent the two most frequent scenarios. Moreover, these two cases are particularly interesting, as they illustrate an intuitive tradeoff between accurate and responsive failure detectors. Firstly, in a failure free execution, a failure detection mechanism can only slow down the consensus algorithm, by generating contention on the resources. So the best solution in the failure free case is no failure detection mechanism at all. Secondly, in the case of one crash, the best solution for the failure free case becomes the worst solution: without a failure detection mechanism the algorithm never terminates! So, there is an obvious trade-off between the two scenarios that we analyze. Moreover, there is a trade-off within the one crash case itself:

- Reducing the termination time in the crash case requires fast reaction to the crash.
- Too many "failure detection" messages increase the contention on the resources (network, CPU), namely, slow down the algorithm. Sadly, fast reaction to process crash requires frequent "failure detection" messages.

In other words, tuning the detection implementation is not an easy task. Among the four implementations described above, we have simulated only the last three ones: it can easily be shown that the algorithm-specific heartbeat implementation of Section 4.4 is better than the general heartbeat implementation of Section 4.1. The results presented below have been obtained by averaging over a large number of simulations (thousands). Confidence intervals for these results have been computed in [13]. All simulations were made for $n = 5$ processes.

5.1. "Interrogation" implementation

The average termination time of the consensus algorithm, using the *interrogation* implementation, is illustrated in Figure 8. For several values of Δ_i, the termination time is plotted as a function of the timeout delay Δ_{to}. Figure 8(a) shows the results obtained for failure free executions. Figure 8(b) depicts the termination time of the algorithm when the coordinator of the first round crashes.

Failure free case. Figure 8(a) shows that the termination time decreases as Δ_i increases. For the four values of Δ_i considered, the optimum timeout value Δ_{to} is 6 ms. For smaller values of Δ_{to}, the probability of erroneous suspicions increases. This in turn increases the number of rounds of the algorithm, and thus its termination time. Figure 8(a) also shows that a value of 15 ms for Δ_i is optimal. In this case, the termination time is almost equal to 15 ms.

[4]For the consensus algorithm, the worst case of a crash occurs if the coordinator crashes at the time it tries to send its proposition. In Figure 1, this occurs in phase 2, when p_1 sends its proposition.

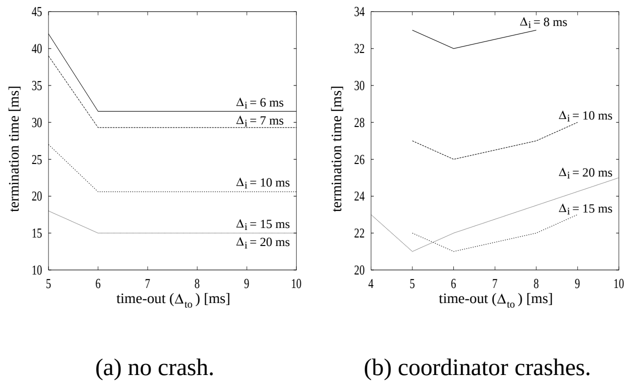


Figure 8. Failure detectors: *interrogation* **implementation.**

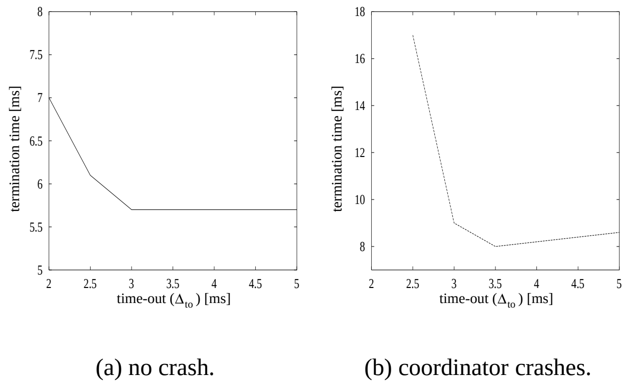


Figure 9. Failure detectors: *algorithm-specific "silent"* **implementation.**

Crash of the coordinator. According to Figure 8(b), it is clear that $\Delta_i = 8$ ms and $\Delta_i = 10$ ms is a bad choice. At first glance, this may seem surprising since a small value of Δ_i should account for a quicker detection of the crash. However, the overhead of an increased number of messages outweighs the benefits of a quicker failure detection. $\Delta_i = 15$ ms is a slightly better choice than $\Delta_i = 20$ ms.

Summing up. In the failure free case, the termination time of the consensus is the same for $\Delta_i = 15$ ms and $\Delta_i = 20$ ms. However, $\Delta_i = 20$ ms leads to a slower failure detection, and thus to an increased termination time in the event of a crash. In conclusion, the optimal choice for the parameters is $\Delta_i = 15$ ms and $\Delta_{to} = 6$ ms. This choice leads to a termination time of 15 ms in the failure free case, and of 21.7 ms for the worst case of one crash.

5.2. Algorithm-specific "silent" implementation

Figure 9 illustrates the termination time of the consensus, using the algorithm-specific "silent" implementation of failure detectors. With this implementation, the failure detectors are characterized by only one parameter: Δ_{to} (see Sect. 4).

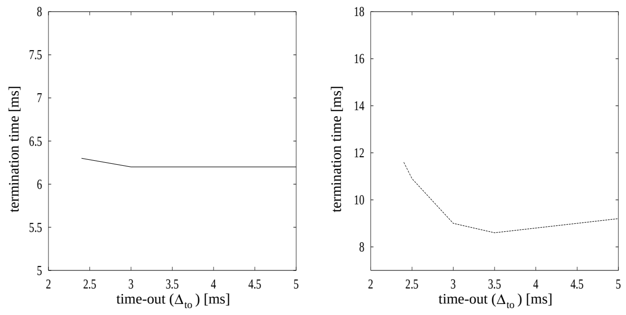
Failure free case. Figure 9(a) depicts the termination time in the failure free case. The figure shows that the termination time of the consensus algorithm is minimized when Δ_{to} is larger than 3 ms: when Δ_{to} is smaller than 3 ms, the termination time increases due to erroneous suspicions.

Crash of the coordinator. The termination time when the coordinator crashes is plotted in Figure 9(b). The figure shows that the termination time reaches its minimum for Δ_{to}=3.5 ms. When Δ_{to} is less than 3.5 ms, the termination time increases due to the number of erroneous suspicions. Conversely, a value of Δ_{to} larger than 3.5 ms accounts for a slower detection of the crash, thus increasing the termination time.

Summing up. The termination time of the consensus is optimized for a timeout value Δ_{to}=3.5 ms. This value leads to a termination time of 5.7 ms in the failure free case, and of 8 ms for the worst case of one crash.

5.3. Algorithm-specific "heartbeat" implementation

The results plotted in Figure 10 illustrate the termination time obtained with the *algorithm-specific heartbeat* implementation of the failure detectors. This simulations have been performed with a value of Δ_i slightly smaller than Δ_{to} ($\Delta_i = 98\%$ of Δ_{to}). Figure 10(a) shows the termination time in the failure free case, whereas the termination time when the coordinator crashes is depicted in Figure 10(b).



(a) no crash. (b) coordinator crashes.

Figure 10. Failure detectors: *algorithm-specific "heartbeat"* **implementation.**

Summing up. This implementation of the failure detectors has a behavior similar to the "silent" implementation (compare Fig. 10 with Fig. 9). A value of $\Delta_{to} = 3.5$ ms allows us to obtain optimal results in both cases. This value leads to a termination time of 6.2 ms in the failure free case, and of 8.6 ms for the worst case of one crash.

5.4. Comparison of the different implementations

Table 1 summarizes the simulation results obtained with the different implementations of failure detectors. The results show that algorithm-specific implementations lead to better performances. This is not surprising: whenever a failure detection mechanism can be designed specifically for an algorithm, the number of messages can be reduced significantly, thus improving the performance. Although the qualitative result is not surprising, a quantification of the differences requires a careful study such as we did here.

The results also show that the *algorithm-specific "silent"* implementation of the failure detectors leads to slightly better results than the *algorithm-specific heartbeat* implementation. This better performance is due to the fine tuning of the failure detection parameters that we did for the special case of $n = 5$ processes. Changing the number of processes would require new simulations to find the optimal parameters for the "silent" implementation. The algorithm-specific heartbeat implementation is in this respect more robust, which is witnessed by a steeper curve in Figure 9(b) than in Figure 10(b), when $\Delta_{to} < 3.5$ ms.

6. Related Work

There have been only few attempts so far at evaluating the impact of a failure detection mechanism on the performance of fault-tolerant algorithms. This is partly due to the fact that many such algorithms are designed with a specific failure detection mechanism in mind, or are based on a group membership service that hides it.

Chen, Toueg, and Aguilera [3] also analyze the efficiency of failure detection mechanisms. They however take a totally different approach to ours, in which they define a set of metrics with which failure detectors can be characterized. For instance, they consider three random variables T_D, T_{MR}, and T_M to represent a failure detectors implementation: The *detection time* T_D represents the time elapsed between the instant when a process p crashes, and the time when p is permanently suspected by some other process q; the *mistake recurrence time* T_{MR} is the time between two consecutive incorrect suspicions; and the *mistake duration* T_M is the duration of incorrect suspicions.

The only practical evaluation of failure detectors that we are aware of is due to Estefanel and Jansch-Pôrto [6]. They propose a highly efficient adaptive heart-beat failure detector algorithm that they evaluate using performance measurements.

These other approaches to the problem of evaluating failure detectors are complementary to our simulations; Chen *et al* take an analytical approach, while Estefanel *et al* opt for performance measurements. In this context, we strongly believe that, combining results obtained using these three different approaches, the evaluation of failure detectors is a required step to understand actual performance issues in fault-tolerant algorithms.

7. Conclusion

The paper has studied the impact of different implementations of failure detectors on the termination time of a consensus algorithm, (1) in failure free executions, and (2) in executions with one process crash. The study has pointed out the trade-off between a short

Table 1. Termination time of consensus (simulation with 5 processes).

failure detection	parameters	failure free execution	coordinator crashes (worst case)
interrogation	Δ_i=15 ms, Δ_{to}=6 ms	15 ms	21.7 ms
specific silent	Δ_{to}=3.5 ms	5.7 ms	8 ms
specific heartbeat	Δ_i=3.4 ms, Δ_{to}=3.5 ms	6.2 ms	8.6 ms

termination time in the failure free case, and a quick reaction to failures. The trade-off explains that finding the "best" implementation of failure detectors is not an easy task. Furthermore, the paper has shown that general implementations of the failure detectors tend to generate unnecessary messages, which has a negative impact on the performance of the consensus algorithm. Specialized implementations lead, on the other hand, to significantly better results since they generate fewer or even no messages.

Finally, this study has shown that (1) implementing failure detectors and (2) designing a consensus algorithm based on a given failure detector, are two orthogonal issues. In other words, it is possible in the context of consensus (and other agreement problems) to decouple timing issues (e.g., implementation of failure detection) from logical issues (i.e., proving the safety and liveness of a specific algorithm based on abstract properties of failure detectors). Such a decoupling, similar to all modular approaches, simplifies the construction and the proof of correctness of complex piece of software. Nevertheless, we have shown that a modular approach does not prevent from considering timing issues when optimal performances are required. In other words, modularity and efficiency are not antagonistic issues.

References

[1] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, July 1996.

[2] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[3] W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. In *Proc. of the IEEE Int'l Conf. on Dependable Systems and Networks (ICDSN-30)*, June 2000.

[4] F. Cristian, R. de Beijer, and S. Mishra. A performance comparison of asynchronous atomic broadcast protocols. *Distributed Systems Engineering*, 1(4):177–201, June 1994.

[5] X. Défago, P. Felber, and A. Schiper. Optimization techniques for replicating CORBA objects. In *Proc. of the 4th IEEE Int'l Workshop on Object-oriented Real-time Dependable Systems (WORDS)*, Jan. 1999.

[6] L. A. B. Estefanel and I. Jansch-Pôrto. On the evaluation of heartbeat-like detectors. In *2nd IEEE Latin-American Workshop*, Cancun, Mexico, Feb. 2001.

[7] C. Fetzer and F. Cristian. Fail-awareness: an approach to construct fail-safe applications. In *Proc. of the 27th IEEE Int'l Symp. on Fault-Tolerant Computing (FTCS-27)*, pp.282–291, June 1997.

[8] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, Apr. 1985.

[9] R. Guerraoui, M. Larrea, and A. Schiper. Reducing the cost for non-blocking in atomic commitment. In *Proc. of the 16th IEEE Int'l Conf. on Distributed Computing Systems (ICDCS-16)*, pp.692–697, May 1996.

[10] R. Guerraoui and A. Schiper. Consensus service: a modular approach for building fault-tolerant agreement protocols in distributed systems. In *Proc. of the 26th IEEE Int'l Symp. on Fault-Tolerant Computing (FTCS-26)*, pp.168–177, June 1996.

[11] N. Malcolm and W. Zhao. Hard real-time communication in multiple-access networks. *Real-Time Systems*, 8:35–77, 1995.

[12] N. Sergent. Evaluating latency of distributed algorithms using Petri nets. In *Proc. of the 5th Euromicro Workshop on Parallel and Distributed Processing*, pp.437–442, Jan. 1997.

[13] N. Sergent. *Soft real-time analysis of asynchronous agreement algorithms using Petri nets*. PhD thesis, EPFL, Switzerland, 1998. Number 1808.

[14] N. Sergent, X. Défago, and A. Schiper. Failure detectors: Implementation issues and impact on consensus performance. TR SSC/1999/019, EPFL, Switzerland, May 1999.

[15] P. Urbán, X. Défago, and A. Schiper. Contention-aware metrics for distributed algorithms: Comparison of atomic broadcast algorithms. In *Proc. of the 9th IEEE Int'l Conf. on Computer Communications and Networks (IC3N 2000)*, Oct. 2000.