# Failure Detectors: implementation issues and impact on consensus performance

Nicole Sergent* Xavier Défago André Schiper

nsergent@vaudoise.ch xavier.defago@epfl.ch andre.schiper@epfl.ch

*Laboratoire de Systèmes d'Exploitation*
*École Polytechnique Fédérale de Lausanne, Switzerland*

**Abstract**

Due to their nature, distributed systems are vulnerable to failures of some of their parts. Conversely, distribution also provides a way to increase the fault tolerance of the overall system. However, achieving fault tolerance is not a simple problem and requires complex techniques.

An agreement problem known as the problem of consensus is at the heart of most problems encountered during the design of a fault tolerant system. This problem is however not solvable in the asynchronous system model, unless the model is augmented with adequate failure detectors. The resulting system model is a time-free model since all timing issues are abstracted by the characteristics of the failure detectors.

It is sometimes claimed that time-based system models are more realistic than time-free models for solving distributed agreement problems. The goal of this paper is to show that solving consensus in the asynchronous system model augmented with failure detectors does not prevent from considering timing issues. We consider the consensus algorithm with various implementations of failure detectors, and we analyse their impact on the termination time of the consensus algorithm.

This study shows that the design of fault-tolerant distributed algorithms in the asynchronous system model augmented with failure detectors is *orthogonal* to the issue of implementing the actual failure detectors. This nicely decouples logical issues (proof of safety and liveness of an algorithm) 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.

*Currently affiliated to *Vaudoise Assurances*.

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 [6]. In a few words, the reason for this impossibility is that it is formally impossible, in a purely asynchronous system, to distinguish a crashed process from a very slow one.

In order to solve consensus, Chandra and Toueg have extended the asynchronous system model with the notion of failure detectors [2]. In this work, they identify families of failure detectors according to their properties. With Hadzilacos, they have shown that the failure detector $\diamond\mathcal{S}$ is the weakest failure detector that allows to solve consensus in asynchronous systems [1]. The work of Chandra and Toueg shows that, given a failure detector with the correct properties, the consensus is solvable in an asynchronous system. But, more importantly, the consensus algorithm for $\diamond\mathcal{S}$ presented by Chandra and Toueg needs a failure detector to ensure *liveness*, but it always ensures *safety*. In other words, if the algorithm is run in a purely asynchronous system, it may never terminate but it will never take wrong decisions.

The goal of the paper is to show that solving consensus in the failure detector model,[1] does not prevent from considering 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 consensus 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 analysing 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 concludes the paper.

# 2 Background

## 2.1 Failure detector model

A failure detector is a local module attached to every process. Its 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,

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

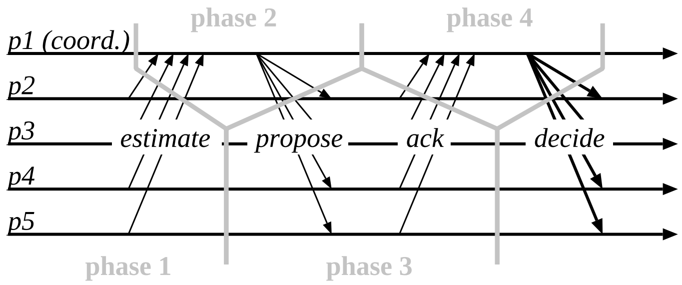



Figure 1: Consensus algorithm when no failure or suspicion occurs ($n = 5$ processes).

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 shown that $\Diamond\mathcal{S}$ is the weakest failure detector class allowing 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].

## 2.2 Consensus

Consensus is a central problem in the context of distributed systems, being at the heart of many agreement problems [2, 8, 7]. The problem is defined on a set $\Pi$ 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.

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 and suspicion free run. The figure shows the four phases that constitute one round. Informally, the algorithm works as follows.

- In the first phase, the processes send their estimate to the coordinator.
- In the second phase, the coordinator waits for a proposition from a majority of the processes and proposes a value chosen among the estimations.
- In the third phase, the processes wait for a proposition from the coordinator. They adopt the value proposed by the coordinator and acknowledgement it (*ack* message) and 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 proceeds to the next round.
- In the fourth phase, 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.

A more detailed description of this algorithm can be found in [2], together with the proofs. Some practical optimizations to this algorithms are presented in [4].

# 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 is 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

The key point in our simulation is the model that we adopt for computing transmission delays of messages. Usual simulation studies compute message transmission delays using some empirical distribution (e.g., [3]). However, such an approach *does not take into account the network contention caused by the studied system 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*. Informally, this model can be described as follows [11].[2] A message $m$ sent from process $p$ on a *sending host* to process $q$ on a *receiving host* requires the allocation of three resources (Fig. 2):

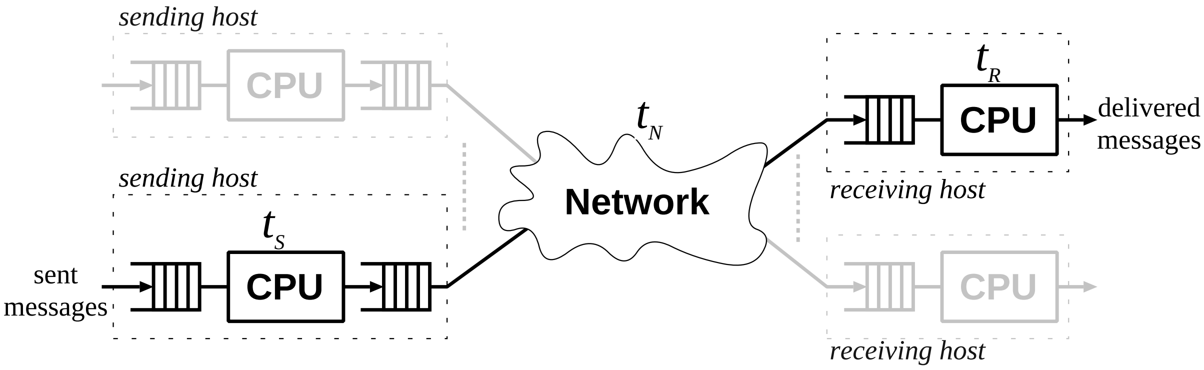



Figure 2: Modelling message transmission

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

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 locally, based on a FIFO policy. The network resource is slightly more complex. For each host, the messages are handed over to the network according to a FIFO policy. However, the access to the network is allocated randomly between those hosts that have a message to send.

We assume only short messages (i.e., messages of a few bytes at the level of the algorithm), and consider that transmitting a message requires the above three resources for the following constant durations.

- 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 [11, 10]. We have then validated our message transmission model using the communication schema of a *two-phase commit* protocol (2PC). We have measured the experimental time for different number of processes and compared these values with the results obtained by simulation. The simulation results lie within 6% of the experimental results [11],[3] which shows that our model is fairly good.

[2] A similar approach has also been used in [9].

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

# 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 ad hoc 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).

To simplify the presentation, we do not follow the model of [2], in which the failure detector $FD_p$ (attached to process $p$) is distinguished from $p$. Here we simply consider that $p$ itself manages the list of processes that it suspects.

## 4.1 "Heart-beat" implementation

*Heart-beat* 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.

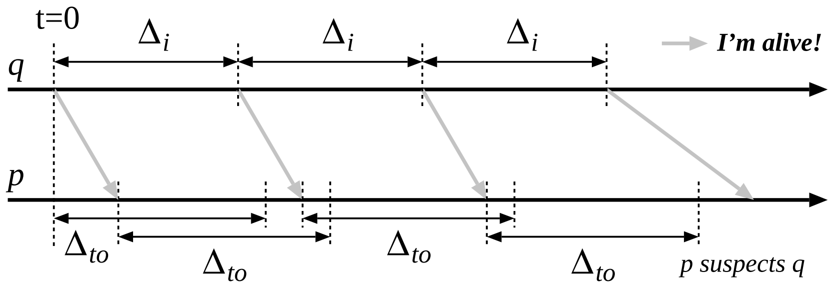



Figure 3: Failure detection: the *heart-beat* implementation.

As illustrated in Figure 3, the failure detector is defined by two parameters: the *heartbeat period* $\Delta_i$ and the *time-out delay* $\Delta_{to}$.

## 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.

With this implementation, the failure detector is also defined by two parameters: the *interrogation period* $\Delta_i$ and the *time-out delay* $\Delta_{to}$ (Fig. 4). Every $\Delta_i$, 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 time-out delay $\Delta_{to}$, 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.

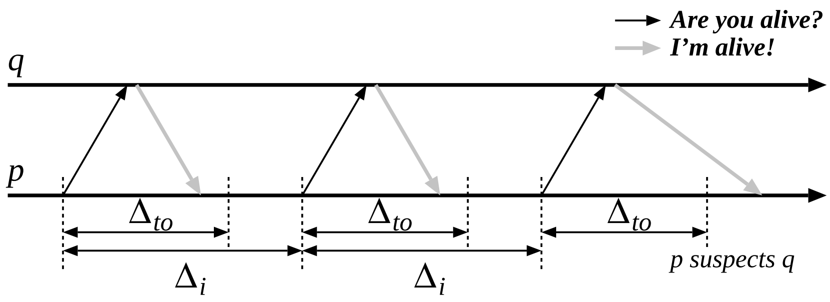



Figure 4: Failure detection: the *interrogation* implementation.

## 4.3 Ad hoc "no message" implementation

Ad hoc implementations of failure detection originate 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.

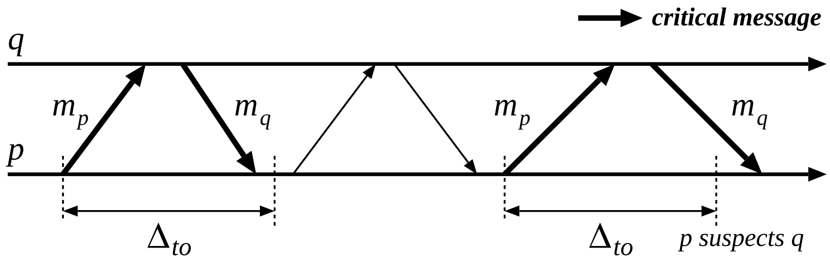



Figure 5: Failure detection: the *"no message"* implementation.

We can abstract the details of the consensus algorithm by introducing the notion of *critical message* and *critical response* (Fig. 5). A critical message and the critical response are messages generated by the algorithm executed by process $p$ and process $q$. 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 way by a simple time-out mechanism, without any additional "failure detection" message. If $p$ does not receive $m_q$ after a delay $\Delta_{to}$, then $p$ suspects $q$. A similar approach is used in [5].

This ad hoc implementation of failure detection is extremely cheap in terms of messages, as it does not generate any "failure detection" message. In the context of the consensus algorithm considered here [2], the critical message/response pattern occurs when $q$ is the coordinator of some round $r$, $m_p$ is the *estimate* message sent by a participant $p$ to the coordinator $q$, and $m_q$ is the *propose* message sent by the coordinator to a participant.

The drawback of the "no message" 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 time-out and incorrectly suspect $q$.

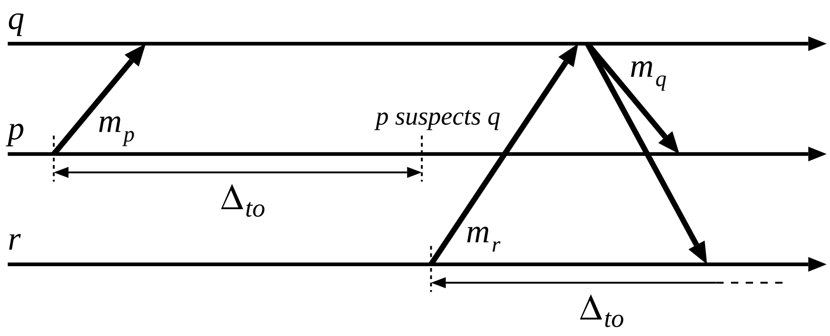



Figure 6: Incorrect suspicions with the "no message" implementation.

## 4.4 Ad hoc "heart-beat" implementation

The *ad hoc heart-beat* implementation overcomes the drawback of the "no message" 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 *heart-beat* messages to that process. The emission of heart-beat messages stops after $q$ has sent its critical response $m_q$ (Fig. 7).

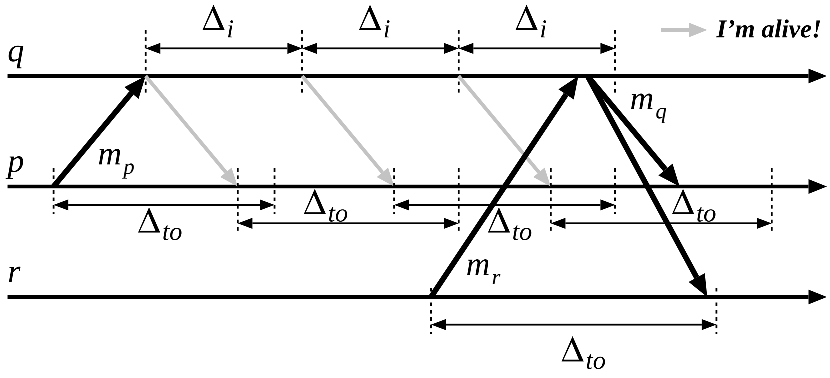



Figure 7: Failure detector: the *ad hoc heart-beat* implementation.

With this implementation, the failure detector is characterized by the two parameters $\Delta_i$ and $\Delta_{to}$. The parameter $\Delta_i$ sets the frequency of the heart-beat messages, while the parameter $\Delta_{to}$ defines the time-out delay.

There are certain obvious restrictions on the choice of $\Delta_i$ and $\Delta_{to}$. Considering Figure 7, $\Delta_{to}$ should be larger than $\Delta_i$ or else the heart-beat cannot be received on time. Furthermore, Figure 7 also shows that $\Delta_{to}$ should be greater than the average round-trip time.

# 5 Impact of failure detectors

In this section, we analyse the impact of the different implementations of failure detection on the termination time of a consensus algorithm, using the simulation model of Section 3. We analyse 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.

1. 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.

2. In the case of one crash, the best solution for the failure free case becomes the worst solution. With no failure detection mechanism the algorithm never terminates!

There is obviously a 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 process crash.

- Too many "failure detection" messages increase the contention on the resources (network, CPU), i.e., slow down the algorithm. Sadly, fast reaction to process crash requires frequent "failure detection" messages.

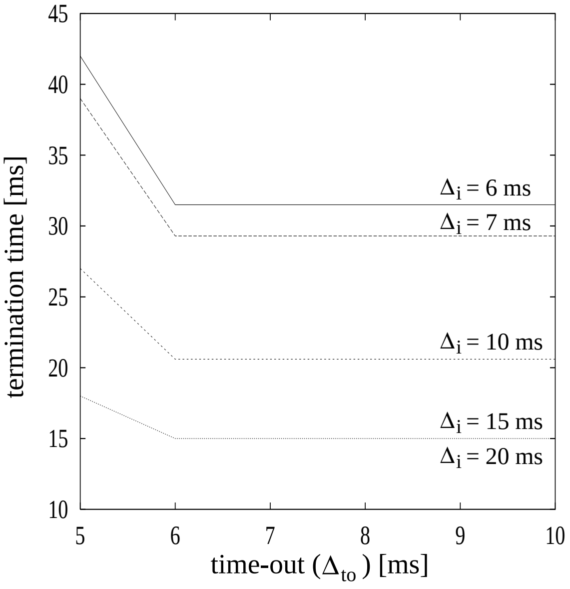
In other words, tuning the detection implementation is not an easy task. Among the four implementations described above, we have simulated only the last three ones: it can easily be shown that the ad hoc heart-beat implementation of Section 4.4 outperforms the general heart-beat implementation of Section 4.1. The results presented were obtained by averaging over a large number of simulations (thousands). Confidence intervals for these results have been computed in [11]. All simulations were made for $n = 5$ processes.

## 5.1 "Interrogation" implementation

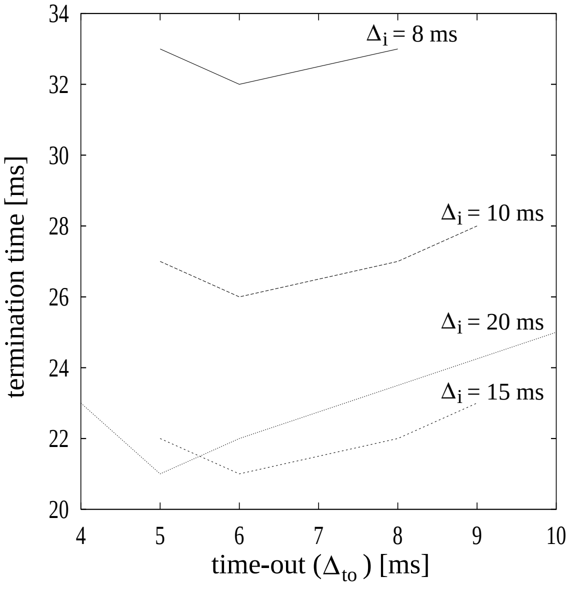
The average termination time of the consensus algorithm, using the *interrogation* implementation, is illustrated in Figure 8. For several values of $\Delta_i$, the termination time is plotted as a function of the time-out delay $\Delta_{to}$. Figure 8(a) shows the results obtained for failure free executions. 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 $\Delta_i$ increases. For the four values of $\Delta_i$ considered, the optimum time-out value $\Delta_{to}$ is 6 ms. For smaller values of $\Delta_{to}$, 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 $\Delta_i$ 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.




(a) no crash.




(b) coordinator crashes.

Figure 8: Failure detectors: *interrogation* implementation.

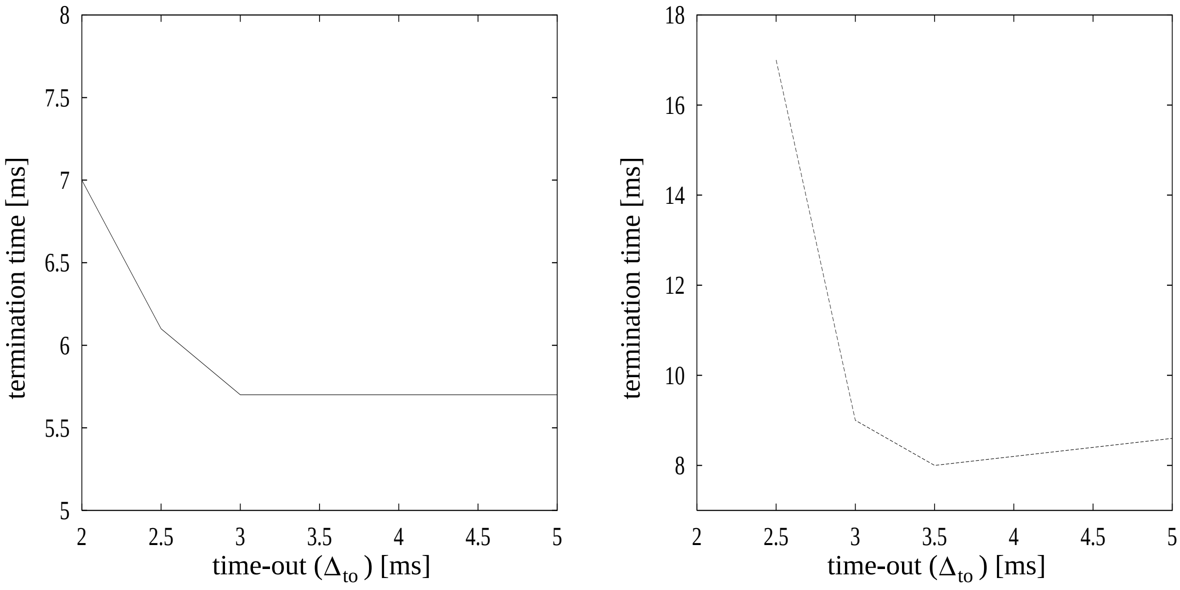
**Crash of the coordinator.** According to Figure 8(b), it is clear that $\Delta_i = 8$ ms and $\Delta_i = 10$ ms is a bad choice. At first glance, this may seem surprising since a small value of $\Delta_i$ should account for a quicker detection of the crash. However, the overhead of an increased number 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 Ad hoc "no message" implementation

Figure 9 illustrates the termination time of the consensus, using the ad hoc "no message" implementation of failure detectors. With this implementation, the failure detectors are characterized by only one parameter, $\Delta_{to}$ (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 $\Delta_{to}$ is larger than 3 ms: when $\Delta_{to}$ is smaller than 3 ms, the termination time increases due to erroneous suspicions.




(a) no crash.

(b) coordinator crashes.

Figure 9: Failure detectors: *no message* implementation.

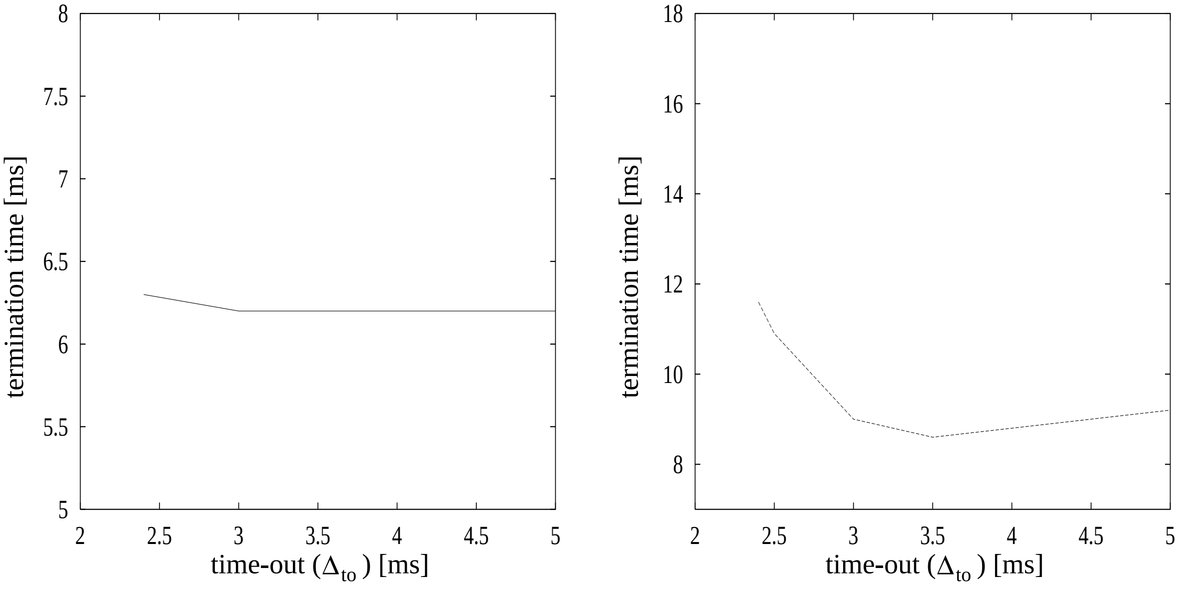
**Crash of the coordinator.** The termination time when the coordinator crashes is plotted in Figure 9(b). The figure shows that the termination time reaches its minimum for $\Delta_{to}$=3.5 ms. When $\Delta_{to}$ is less than 3.5 ms, the termination time increases due to the number of erroneous suspicions. Conversely, a value of $\Delta_{to}$ 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 time-out value $\Delta_{to}$=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 Ad hoc "heart-beat" implementation

The results plotted in Figure 10 illustrate the termination time obtained with the *ad hoc heart-beat* implementation of the failure detectors. This simulations have been performed with a value of $\Delta_i$ slightly smaller than $\Delta_{to}$ ($\Delta_i$ = 98% of $\Delta_{to}$). 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).

**Summing up.** This implementation of the failure detectors has a behavior similar to the "no message" 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.




(a) no crash.

(b) coordinator crashes.

Figure 10: Failure detectors: *ad hoc heart-beat* implementation.

## 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 ad hoc implementations lead to better performances than general implementations. This is not really a surprise: 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.

| failure detection | parameters | failure free execution | coordinator crashes (worst case) |
|---|---|---|---|
| interrogation | $\Delta_i$=15 ms, $\Delta_{to}$=6 ms | 15 ms | 21.7 ms |
| no message | $\Delta_{to}$=3.5 ms | 5.7 ms | 8 ms |
| ad hoc heart-beat | $\Delta_i$=3.4 ms, $\Delta_{to}$=3.5 ms | 6.2 ms | 8.6 ms |

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

The results also show that the *ad hoc "no message"* implementation of the failure detectors leads to slightly better results than the *ad hoc heart-beat* 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 no message implementation. The ad hoc heart-beat 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 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 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.

Although the quantitative results obtained on a specific consensus algorithm are unlikely to be applicable to other consensus algorithms, it is likely that a study of other algorithms leads to similar qualitative conclusions.

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. *Journal of the ACM*, 43(4):685–722, July 1996.

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

[3] 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.

[4] X. Défago, P. Felber, and A. Schiper. Optimization techniques for replicating CORBA objects. In *Proceedings of the 4th Int'l Workshop on Object-oriented Real-time Dependable Systems (WORDS'99)*, Santa Barbara, CA, USA, January 1999.

[5] C. Fetzer and F. Cristian. Fail-awareness: An approach to construct fail-safe applications. In *Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS-27)*, pages 282–291, Seattle, WA, USA, June 1997. IEEE, IEEE Computer Society Press.

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

[7] R. Guerraoui, M. Larrea, and A. Schiper. Reducing the cost for non-blocking in atomic commitment. In *Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS-16)*, pages 692–697, Hong Kong, May 1996.

[8] R. Guerraoui and A. Schiper. Consensus service: A modular approach for building fault-tolerant agreement protocols in distributed systems. In *Proceedings of the 26th International Symposium on Fault-Tolerant Computing (FTCS-26)*, pages 168–177, Sendai, Japan, June 1996.

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

[10] N. Sergent. Evaluating latency of distributed algorithms using Petri nets. In *Proceedings of the 5th Euromicro Workshop on Parallel and Distributed Processing*, pages 437–442, London, UK, January 1997.

[11] N. Sergent. *Soft Real-Time Analysis of Asynchronous Agreement Algorithms Using Petri Nets*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 1998. Number 1808.