Research Report 182-7/2006 (Journal Version)
Technische Universität Wien, Institut für technische Informatik E182

Optimal Message-Driven Implementations of Omega with Mute Processes

Martin Biely Josef Widder

Abstract

We investigate the complexity of algorithms in message-driven models. In such models, events in the computation can only be caused by message receptions, but not by the passage of time. Hutle and Widder [2005a] have shown that there is no deterministic message-driven self-stabilizing implementation of the eventually strong failure detector and thus Ω in systems with uncertainty in message delays and channels of unknown capacity using only bounded space. Under stronger assumptions it was shown that even the eventually perfect failure detector can be implemented in message-driven systems consisting of at least $f + 2$ processes (f being the upper bound on the number of processes that crash during an execution).

In this paper we show that $f+2$ is in fact a lower bound in message-driven systems, even if non-stabilizing algorithms are considered. This contrasts time-driven models where $f+1$ is sufficient for failure detector implementations.

Moreover, we investigate algorithms where not all processes send message, i.e., are active, but some (in a predetermined set) remain passive. Here, we show that the $f + 2$ processes required for message-driven systems must be active, while in time-driven systems it suffices that f processes are active.

We also provide message-driven implementations of Ω. Our algorithms are efficient in the sense that not all processes have to send messages forever, which is an improvement to previous message-driven failure detector implementations.

This research report also appears in the Association for Computing Machinery (ACM) Transactions on Autonomous and Adaptive Systems (TAAS), a short version appeared in the Proceedings of the Eighth International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2006) in Lecture Notes in Computer Science (LNCS) vol. 4280.
Josef Widder is partially supported by the Austrian FWF project *Theta* (proj. no. P17757). Martin Biely is supported by the Austrian BM:vit FIT-IT project *TRAFT* (proj. no. 812205).
Authors' present addresses: Martin Biely, Vienna University of Technology, Embedded Computing Systems Group (E182-2), Treitlstraße 3/2, A-1040 Wien. Josef Widder, Laboratoire d'Informatique, Ecole Polytechnique, Palaiseau, France.

1 Introduction

When building autonomous systems, one has to conceive the components such that they can survive long periods of time without human intervention, leading to the requirement of built-in fault tolerance. Moreover, the distributed components of the autonomous system have to work together leading to the requirement of coordination and agreement. Reaching agreement in the presence of faults [Pease et al. 1980] is a central problem of distributed computing.

In the area of agreement problems, it was shown in [Fischer et al. 1985] that consensus — the problem of agreeing on a common value despite faults [Pease et al. 1980; Lynch 1996] — cannot be solved deterministically in systems without assumptions on the timing behavior of message transmissions or assumptions on relative computing speeds. Consequently, in the classic literature on fault-tolerant agreement problems, specifically the role of time — or rather synchrony — is heavily researched [Dolev et al. 1987; Dwork et al. 1988; Santoro and Widmayer 1989; Chandra and Toueg 1996]. In this context, synchrony encapsulates assumptions on the timing behavior of components as upper and lower bounds on message delays and relative computing speeds. It can be assumed that there are such bounds that hold in all executions (termed "known" bounds in [Dwork et al. 1988]) or it can be assumed that in every execution there is some (execution specific) bound (called unknown bound).

As a result of research on consensus, it was shown that if just crash faults are contemplated, synchrony can be encapsulated by (unreliable) failure detectors [Chandra and Toueg 1996] and that the eventual leader oracle Ω is the weakest failure detector (FD) to allow solving consensus [Chandra et al. 1996]. Intuitively, Ω is a distributed oracle that provides processes with the name of a process guaranteeing that eventually all processes will be provided with the name of a unique correct process. Obviously, implementing Ω presents a problem of its own that can be solved with synchrony assumptions, such that much work focused on contemplating timing models to that end and in fact very weak models have been established [Aguilera et al. 2003; Malkhi et al. 2005; Hutle et al. 2006; Hutle et al. 2008]. However, as indicated above, timing is not the sole parameter of distributed computing models: others are for example atomicity of events (e.g., broadcast vs. unicast) and event generation.

In this paper we do not focus on the timing of an execution but we investigate the orthogonal issue of how the events that constitute a distributed computation are triggered. Here we distinguish two possibilities: time-driven and message-driven models. In the classic time-driven models [Fischer et al. 1985; Dolev et al. 1987; Dwork et al. 1988; Chandra and Toueg 1996], events are triggered locally by the passage of time, i.e., by clocks or timers. Conceptually, this can be seen as Turing machines or state machines interconnected by a network, where state transitions can be enabled by the passage of time.

In the more recent message-driven execution models [Hutle and Widder 2005a; Robinson and Schmid 2008; Widder et al. 2005; Fuegger et al. 2006], events — i.e., state transitions — can only occur as immediate reaction to the reception of a message. Therefore, message-driven algorithms do not need to have access to a clock. In other words, in message-driven executions, processes only perform computations when necessary. (This might be relevant for systems where low power consumption is essential, e.g. wireless sensor networks.) Algorithms that perform steps only upon message receptions were discussed in [Fischer and Lamport 1982] and named "asynchronous" there (similar to asynchronous network protocols that operate by explicit handshaking).

Given the generality of the above mentioned Turing machines, and thus timed models, it might appear that message-driven models are artificially restrictive when considering the traditional systems under consideration in distributed computing research, i.e., computers linked by a network. It is nevertheless the case that message-driven models appear to be the natural choice when considering new application domains for distributed algorithms like the design of asynchronous (delay insensitive [Ebergen 1991]) circuits. Asynchronous hardware differs from synchronous one in how computations are triggered. In synchronous designs, there exists a central clock whose ticks trigger each component (conceptually) simultaneously. The inputs of a component are ready when the tick signal reaches the component, and its outputs are present at the next stage of the logic at the next tick. Thus, the time between two ticks must be aligned to worst case switching times of the slow components, s.t. synchronous circuit designs have the same properties as lock-step synchronous distributed system models which are often "impossible or inefficient to implement [...] in many types of distributed systems" [Lynch 1996, page 5].

The concept of periodically executed steps is not appropriate for some applications, and for asynchronous chip design (due to the lack of a central clock) in particular. Fuegger et al. [2006] showed how a message-driven algorithm can be implemented in asynchronous hardware. It was shown that the central clock can be replaced by a VLSI implementation of a message-driven fault-tolerant clock generation algorithm. Consequently, there is a requirement to understand possibilities and restrictions of the message-driven model both for building real systems and for understanding the very nature of "synchrony".

Most of the algorithms that solve consensus or implement a failure detector — although often presented in a message-driven style — are based on time-driven execution models, i.e., steps can be taken whether messages are in the incoming buffers or not.[1] Only little work exists in the context of consensus that considers how events in a distributed computation are triggered. For example, interrupt style failure detectors and their expressive power are discussed in [Gärtner and Pleisch 2001]. Hutle and Widder [2005a] have shown that there is a difference between the expressiveness of time-driven and message-driven execution models, by showing that Ω cannot be implemented with a self-stabilizing message-driven protocol under certain assumptions where self-stabilizing time-driven protocols [Beauquier and Kekkonen-Moneta 1997] can solve the problem. By strengthening the assumptions, it was shown that the problem has message-driven implementations as well.

These message-driven and time-driven solutions differ, however, in the number of processes required. The aforementioned message-driven FD implementations require systems of at least $f + 2$ processes — f being the upper bound on the number of processes that may fail during an execution (by crashing). Time-driven implementations of Ω are known that require just $f + 1$ processes, therefore the question arises whether the algorithms in [Hutle and Widder 2005a] are optimal regarding the required number of processes and thus whether the choice of message-driven vs. time-driven algorithms has consequences with respect to the complexity and resilience of solutions for non stabilizing systems. In this paper we answer this question in the affirmative. We present an $f+2$ lower bound on the number of active processes required for message-driven implementations of Ω.

Our result does of course also apply to "stronger" failure detectors, i.e., those from which Ω can be extracted. This is obviously also true for perpetual failure detectors, such as the perfect failure detector.

Additionally, we consider efficiency issues: It is known that for time-driven systems, communication efficient failure detector implementations are possible [Aguilera et al. 2004]. Here, communication efficiency refers to the number of processes that have to keep sending messages forever, or the number of links that have to carry messages during the whole execution. All known message-driven failure detector implementations [Hutle and Widder 2005a; Le Lann and Schmid 2003; Widder et al. 2005; Hermant and Widder 2005] require all correct processes to keep sending messages forever. We show that this is not necessary for implementing Ω. Our algorithms require at most $f + 2$ processes to send messages while the other ones may remain mute. Our lower bound theorem does also hold for such communication efficient algorithms, that is, our algorithms are optimal in this respect.

This reduced complexity may not seem particularly relevant at first sight, but one remarkable fact is that for solving consensus in the asynchronous

[1]Distributed algorithms often include statements like "if $f + 1$ messages m have been received then send message m' to all". However, in time-driven systems, it is not the arrival of the message m which triggers the sending of m', but some separate computing step which could in fact happen (much) later in the execution.

system augmented with Ω one requires $2f+1$ processes, while — with our algorithm — only $f+2$ have to actively participate in doing failure detection. It follows that only in the case where the number of processes $N=3$ and $f=1$ all processes have to send messages. In all other systems in which consensus can be solved, our implementation of Ω allows $N-f-2 > \frac{N}{2}-2 \geq 0$ processes to remain silent.

1.1 Contribution

This paper establishes a lower bound on the number of processes required to solve problems in the area of fault-tolerant distributed computing with message-driven protocols. To this end, we employ a rather conservative synchrony assumption in order to make explicit the peculiarities of message-driven models. Even under our strong assumptions, the difference in the lower bound follows — although time-driven protocols are known that implement Ω with $f+1$ processes even under far less restrictive synchrony assumptions [Aguilera et al. 2004].

We also present algorithms that show that this bound is tight. Additionally, the algorithms allow some processes to remain silent throughout the execution. As discussed above, synchrony is a very fundamental property. In order to investigate how message-driven algorithms utilize timing assumptions, we introduce two different algorithms, one which establishes a "local" timebase within a subset of the processes (which might in practice correspond to base station in mobile networks) while the second algorithm implements a global timebase built atop stronger synchrony assumptions.

Although we restrict ourselves in this paper to investigate the complexity of implementing Ω, our results can be used to conduct a more general comparison of time-driven and message-driven models: If one assumes upper and lower bounds on message self-receptions (i.e., messages sent from some process to itself), it might appear trivial to simulate (partially synchronous) time-driven executions [Dwork et al. 1988]. It has been shown in [Hutle and Widder 2005a], however, that this is not always possible. If self-stabilization is required, then, informally, the messages that are in transit when faults stop occurring may compress the perceived time in a way that prevents processes from exploiting the synchrony guarantees provided by the system.

The aforementioned simulation is in fact based on the assumption that message transmission delays between two processes are related to self-reception times (in order to ensure e.g. the parameter Δ assumed in [Dwork et al. 1988]). As self-delivery is typically delivered instantly to the sender itself, ensuring such a relation might be unrealistic or expensive, e.g., in the case of VLSI systems [Fuegger et al. 2006]. Thus, we investigate the case where such a relation cannot be ensured, which we model by the absence of a lower bound on the time to locally transfer a message.

1.2 Road map

In the following section we introduce our model which we use to show our lower bound theorem in Section 3. In Section 4, we present an efficient implementation of Ω that is optimal with respect to the derived resilience bound. Another optimal algorithm that establishes a global time base to implement Ω is introduced in Section 5. Section 6 compares the two algorithms with other message-driven FD implementations before we end with some concluding remarks.

2 Model

We consider a system consisting of N distributed processes, which run on a number of processors connected by a communication network. We assume the existence of a reliable (logically) fully connected[2] message-passing network between the processes. Every process has a unique name out of the set $\{1, 2, \ldots, N\}$. The set of all processes will be denoted $\Pi = \{p \mid 1 \leq p \leq N\}$.

The processes perform a distributed computation which proceeds by the computational steps of processes in Π. Since we consider only message-driven computations, a computational *step* is either the initial step (by which the computation is started at every process) or a *message reception step*. In a message reception step a process must receive at least one message, performs a local computation, and may send zero or more messages. The initial step is the only step a process can ever take without receiving a message, and consists only of a local computation and sending of messages.[3]

Processes may fail by permanently crashing, i.e., they do not take any steps after they have crashed. More precisely, the behavior of a faulty process p is described by the fact that p only takes finitely many steps in an execution although infinitely many messages are sent to p during this execution. A process that does not crash in an execution is called *correct* in this execution. At most f out of the N processes in Π may fail during an execution.

One important aspect of our work is that some processes do never send messages but remain mute throughout the execution. They only passively observe the active part of the system which provides them with some perception of synchrony by sending messages. We denote by $\Lambda = \{p \mid 1 \leq p \leq n\}$, with $n \leq N$, the set of processes which constitute the active part of the system. The passive part — i.e., the set of silent processes — will be called $\Sigma = \Pi - \Lambda$. We define $s = |\Sigma| \geq 0$ and since $N = |\Pi|$, we have $N = n + s$.

[2]We add this assumption for strengthening our impossibility result, while for our algorithms there exists links over which messages are never sent.

[3]The initial step s need not be the first step of a process, if it receives a message before s occurred. For the ease of presentation we thus ignore the booting problem, but we note that it can be solved in message-driven systems [Widder 2003].

For our algorithm we assume that $n = f + 2$ and in our lower bound result in Section 3 we show that this is optimal. Further, for the algorithm analysis, we assume the existence of a global Newtonian real-time clock. The processes, however, do not have a way to access this clock, neither do they have any other means of measuring the passage of time locally. This does not only imply the absence of local clocks but also that there are no lower or upper bounds that only restrict the time it takes to perform a computational step — these times are accounted for in the end-to-end delays (read on).

2.1 Timing

Every communication network is bound to cause delays between the time a message is sent in some step by some process p and the time when it causes a message reception step at some other process $q \neq p$. We assume for our algorithm the existence of some unknown upper and lower bounds on these end-to-end delays, i.e., time to transmit and queue the message (at both ends) plus the time to process the message. We denote by $\tau^+ < \infty$ the upper and by $\tau^- > 0$ the lower bound on the end-to-end delay between processes p and q with $p \neq q$, where $p, q \in \Lambda$; that is, the delays of all links involving a process $s \in \Sigma$ just have to be finite. Only for our second algorithm in Section 5 we assume that also the links connecting processes in Λ to those in Σ obey the bounds τ^+ and τ^-. This is necessary as our second algorithm uses a global time base which can only be implemented given global synchrony assumptions. Since our algorithm does not use the links within the set Σ at all, these can be asynchronous (or even be missing).

For our algorithms we do not assume a priori knowledge of τ^+ and τ^-, i.e., we assume that in every execution there are some bounded τ^+ and τ^- (which may be different in different executions). Indeed the knowledge of the values would be useless as processes cannot measure time locally with clocks. Instead we assume the knowledge of the ratio $\Theta = \tau^+/\tau^-$, i.e., there is a finite Θ which holds in every execution. In our analysis we also use the transmission uncertainty $\varepsilon = \tau^+ - \tau^-$.

It has been shown in [Widder 2004] that algorithms designed for this model also work in a model where no bounds on end-to-end delays exists, while just the ratio between the delays of messages concurrently in transit is bounded by some Θ. Informally, τ^+ and τ^- may change during the execution, as long as their ratio continues to be bounded by Θ. Note that Θ — although being dimensionless itself — is in fact the only (time-related) value that processes can observe. Robinson and Schmid [2008] have shown that this assumption can further be weakened without affecting the correctness of algorithms.

Until now we have assumed that $p \neq q$ for a message sent from p to q. For the other case, i.e., self-receptions, we only assume that the transmission is reliable (in order to strengthen our lower bound result in Section 3) while we do not assume any bounds on transmission times except that they are finite. In fact, transmission delays may as well be 0 (which would model writing the message into memory directly instead of sending it over the network). Our algorithms, however, do not use self-receptions.

2.2 Event Generation

For our lower bound (Section 3) we require the following definitions to discriminate between different types of steps. Let p be some process that receives a set M of messages in a message reception step. If all messages in M were sent by p, we call the step a *self reception step*. If at least one message in M was sent by some process $q \neq p$ then the step is called *extrinsic reception step*.

We further need the following definitions: An extrinsic reception step of message m at some process p, where it is locally impossible for p to determine that m was not the last message received by p in the execution is called *potentially final extrinsic reception step*. By *potentially final non self reception step* we denote all steps that are either a potentially final extrinsic reception step or an initial step.

Note that a potentially final extrinsic reception step occurs at p when p has received all messages that causally precede [Lamport 1978] the message that caused the step. In the case of $f + 1$ active processes, it could be the case that there is only one surviving process p, and no message except those sent by p will ever be received by p after this event.

2.3 Failure Detectors

We consider two kinds of failure detectors in this paper, both of which will only output failure information about processes in Λ. The failure detector Ω [Chandra et al. 1996] outputs a single process (its leader estimate), which eventually must be the same correct process at all processes. The formal definition of the eventual leader oracle Ω reads as follows.

(EL) *Eventual Leadership*. There is a time after which all the correct processes always trust the same correct process.

Additionally we will consider a variant of a stronger FD, i.e., the perfect FD $\mathcal{P}$. It was defined [Chandra and Toueg 1996] to fulfill the following two properties:

(SC) *Strong Completeness*. Eventually, every process that crashes is permanently suspected by every correct process.

(SA) *Strong Accuracy*. No process is suspected before it crashes.

As mentioned above, the FDs considered in this paper only output information about Λ. Therefore we define the following generalization of the perfect FD. Thus we define $\mathcal{P}_\Lambda$ via the following two properties:

(LSC) *Limited Strong Completeness.* Eventually, every process $p \in \Lambda$ that crashes is permanently suspected by every correct process $q \in \Lambda$.

(LSA) *Limited Strong Accuracy.* No process $p \in \Lambda$ is suspected by any process $q \in \Lambda$ before it crashes.

Guerraoui and Schiper [1996] introduced Γ-accurate FDs which are similar to $\mathcal{P}_\Lambda$. $\mathcal{P}_\Lambda$, however, restricts both accuracy *and* completeness to some fixed subset Λ of all processes while Γ-accurate FDs only restrict accuracy properties to some fixed subset Γ.

We do not restrict the semantics of the algorithms that use our FDs, i.e., classic query based execution models [Chandra and Toueg 1996] can be employed as well as interrupt based models; discussions on the respective expressiveness can be found in [Gärtner and Pleisch 2001]. If the whole distributed computation (FDs and applications) should be message (i.e., interrupt) driven as described in our model, the FDs have to be an additional source of events for the application. That is, in addition to message reception steps, applications can also take steps whenever the output of the FD — the leader estimate in case of Ω — changes.

3 Lower Bound on the Number of Active Processes

We show that it is impossible to implement Ω with a message-driven algorithm when only $n = f + 1$ processes are active.

The proof of the following theorem is done by contradiction. We will assume that there exists an implementation $\mathcal{I}$ of Ω. In the following, we show how $\mathcal{I}$ must behave if $n-1$ processes in Λ crash during an execution. Then we consider executions where just $n-2$ processes in Λ crash. By indistinguishability to the first execution, $\mathcal{I}$ violates the properties of Ω thus providing the required contradiction.

Theorem 3.1 *There exists no correct message-driven implementation of* Ω *if* $n \leq f + 1$.

Proof: Assume by contradiction that there exists a message-driven implementation $\mathcal{I}$ of Ω, for a system where $n \leq f + 1$.

Let $\mathcal{E}_1$ be the set of all executions of $\mathcal{I}$ where $n-1$ processes in Λ crash. In all these executions there is a final extrinsic reception step or (at least) the initial step at the sole correct processes $p \in \Lambda$ such that there must be at least one potentially final non self reception step at p after which p takes a possibly infinite number of self reception steps. By (EL), however, after some finite number $\ell \geq 0$ of self reception steps, p must set $leader_p = p$ or $leader_p = r \in \Sigma$ permanently — in both cases the leader estimate of p satisfies (EL).

Let $\mathcal{E}_2$ be the set of all executions of $\mathcal{I}$ where $n-2$ processes in Λ crash initially and there are two correct processes $p, q \in \Lambda$, $p \neq q$. Note that just $n-2 < f$ processes in Λ are faulty such that there can be faulty processes also in Σ in $\mathcal{E}_2$. Let further all executions in $\mathcal{E}_2$ be such that all message end-to-end delays between the processes p and q are equal. From this timing behavior it follows directly that all extrinsic reception steps are potentially final extrinsic reception steps as causally dependent events are perceived in temporal order.

We now consider finite prefixes of $\mathcal{E}_2$. Let these finite executions $\mathcal{E}_2'$ have some potentially final non self reception step s at some process p as their final step. For p, every prefix $e \in \mathcal{E}_2'$ is indistinguishable from some finite prefix execution $e_1 \in \mathcal{E}_1$ that is identical to e except that either (1) q crashes in e_1 directly after sending the message that is the cause of s at p, if s is a potentially final extrinsic reception step, or (2) q is initially crashed in e_1, if s is the initial step at p. As there are no synchrony assumptions on self receptions (which allows even zero delays), we can construct a finite execution e' by extending e with $\ell \geq 0$ self reception steps. By indistinguishability of e' to execution e_1, p must set $leader_p = p$ or $leader_p = r \in \Sigma$ for some ℓ. This constructive argument can be applied to every potentially final non self reception step at any of the two correct processes in Λ, such that these processes p and q have to set their leader estimate as described above.

Since, by (EL), all correct processes must permanently trust one process, $v \in \{p, q\}$ must either set $leader_v = v$ or $leader_v = r \in \Sigma$ upon every (following) potentially final extrinsic reception step. It follows that they cannot set $leader_p = p$ and $leader_q = q$ permanently, as this would violate the "the same correct process" requirement. Thus, $v \in \{p, q\}$ must set $leader_v = r \in \Sigma$. If $|\Sigma| = 0$, we have already reached a contradiction since p and q cannot reach the same leader estimate. If $|\Sigma| > 0$, we observe that only less than f processes in Λ crash in all $\mathcal{E}_2$ executions. That is, at least one process in Σ can crash in such executions. Since r is in Σ, it never sends messages such that p and q cannot distinguish executions where r is correct from ones where r crashes. It follows that there exist executions where permanently $leader_p = r$ but r is crashed which violates (EL). We again reach a contradiction. $\square$

Corollary 3.2 *There exists no correct message-driven implementation of Ω if $n \leq N \leq f + 1$.*

Corollary 3.2 shows that the self-stabilizing algorithms in [Hutle and

Widder 2005a; Hutle and Widder 2005b] are optimal regarding the number of processes required. The impossibility of [Hutle and Widder 2005a], however, even holds if there are synchrony assumptions on self-receptions, whereas the proof of Theorem 3.1 is based on the absence of synchrony assumptions for self-reception. More precisely the absence of a *lower bound* is used in order to be able to extend the executions $\mathcal{E}_2'$.

Note that self-stabilization was not used in the proof of Theorem 3.1, such that the complexity gap is due to the difference in the expressiveness of message-driven respectively time-driven models (and not due to self-stabilization).

Executions of algorithms, where active processes never send messages to themselves obviously do not include self reception steps. Consequently, we can apply the proof of Theorem 3.1 with $\ell = 0$ to arrive at the following observation.

Observation 3.3 *There exists no correct message-driven implementation of Ω where any active process p sends messages only to processes $q \neq p$ if $n \leq N \leq f + 1$.*

Since algorithms of this paper do not create self receptions, Observation 3.3 shows that our algorithms are optimal as well.

Having established the lower bound of Theorem 3.1 we now shortly discuss the relation of active and passive processes in implementations of Ω. Note carefully that if during the course of an execution, f active processes are reliably detected to have crashed, all the remaining processes must be correct as the allowed number of crashes is reached. Thus it is possible to elect an arbitrary correct — and thus even mute — process as leader: Assuming, for instance, that process names are totally ordered, the process with the minimal name among the correct processes may be chosen.

Our lower bound can therefore be interpreted in such a way that $f + 1$ active processes are not sufficient to reliably detect that f active processes have crashed in message-driven systems (due to the indistinguishability of the executions $\mathcal{E}_1$ and $\mathcal{E}_2$). In contrast, for time driven systems it suffices to have f active processes: Assume a lock-step synchronous time-driven system [Lynch 1996], where the f processes with the smallest names are the active ones. They simply send out ping messages in every round and all processes elect their leader by choosing the process with the lowest identifier that they consider to be alive (similar to the implementation of the perfect failure detector in [Lynch 1996, page 796]). Here, the crucial observation is that as long as not all the f active processes have crashed, there is always one active process which can be elected, since all process consider it to be alive due to its pings. When all f active processes have crashed by some round r, all remaining (silent) processes know at least from round $r + 1$ on that all silent processes have to be correct and can safely elect the mute

Algorithm 1 Failure Detector Implementation

```
Code for processes $p \in \Lambda$:
1: $phase_p \in \{0, 1\} \leftarrow 0$
2: $leader_p \in \Lambda \leftarrow \min_r\{r \in \Lambda\}$
3: $suspects_p \subset \Lambda \leftarrow \{\}$
4: $\forall q \in \Lambda :\ lastmsg_p[q] \in \{0, \dots, \Phi\} \leftarrow 0$

5: upon initialization do
6:     send $(p, phase_p, 1)$ to $\Lambda$

7: upon reception of $(p, ph, k)$ from $q$ do
8:     if $ph = phase_p$ and $k > lastmsg_p[q]$ then
9:         $lastmsg_p[q] \leftarrow k$
10:        if $k < \Phi$ then
11:            send $(p, phase_p, k + 1)$ to $q$
12:        else
13:            $suspects_p \leftarrow \{r \mid r \in \Lambda \wedge lastmsg_p[r] = 0\}$
14:            $leader_p \leftarrow \min_r \{r \mid r \in (\Lambda - suspects_p)\}$
15:            $phase_p \leftarrow 1 - phase_p$
16:            $\forall r \in \Lambda :\ lastmsg_p[r] \leftarrow 0$
17:            send $(p, phase_p, 1)$ to $\Lambda$
18:            if $p = leader_p$ then
19:                send $(p)$ to $\Sigma$

20: upon reception of $(q, ph, k)$ from $q$ do
21:     send $(q, ph, k)$ to $q$

Code for processes $p \in \Sigma$:
22: $leader_p \in \Lambda \leftarrow \min_r\{r \in \Lambda\}$
23: upon reception of $(q)$ from some $q \in \Lambda$ do
24:     $leader_p \leftarrow q$
```

process with the smallest name. We thus see that in time-driven systems f active processes are sufficient, while in message-driven systems even $f + 1$ are not.

In Section 4 we will establish that in fact $f + 2$ active processes are necessary and sufficient in message-driven systems to implement Ω and thus show that not only there is a difference in the number of processes but also in the number of active processes.

4 A Matching Algorithm

The algorithm has different code for the processes of Λ and Σ. The code for processes $p \in \Lambda$ is a variant of the bounded memory algorithm of [Hutle and Widder 2005a]. Each active process p exchanges messages of type (p, ph, k) with the other processes in Λ, where ph is the phase number in $\{0, 1\}$ and k is an integer that is increased with every round trip. When an active process q receives such a message, q just returns it to p (`line 21`). For

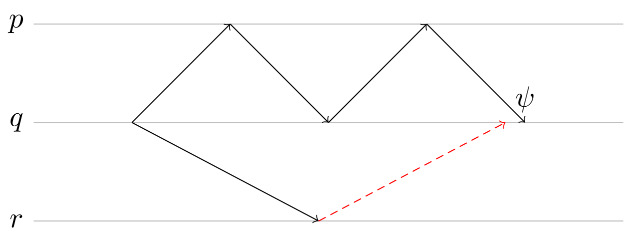

Figure 1: Example of $\Theta < 2$ where the absence of a message from r at event ψ indicates that r must have crashed.

all $q \in \Lambda$, p holds a variable $lastmsg_p[q]$, where it stores the highest integer k received in a (p, ph, k) reply from q. When such an integer k reaches Φ for some phase ph, a new phase $1 - ph$ is started and for all q, $lastmsg_p[q]$ is reset to 0 by process p. If Φ is chosen properly, it is ensured that the minimal duration of a phase is larger than the time required for a round-trip and thus no (p, ph, k) messages are in transit, when process p starts a new phase with phase number ph (after it has terminated phase $1 - ph$). Thus p measures "time" by this message pattern such that p can correctly suspect a process q of being crashed upon termination of Φ round trips if there was no round trip terminated by q (cf. Figure 1).

The code for processes $q \in \Sigma$ simply sets the leader upon reception of an estimate sent by some $p \in \Lambda$. We assume that eventually all messages sent over links from process $p \in \Lambda$ to process $q \in \Sigma$ are received after some finite time (no message loss). Trivially, this algorithm works also in systems where links between processes in Λ and Σ also obey the Θ assumption (cf. Section 2.1).

In contrast to the ideas used for the silent processes in the algorithm in Section 5, this is a very generic method. As long as only the leader announces itself infinitely often to silent processes, it does not matter which algorithm is used to elect the leader among the active processes.

Before we establish the first failure detector property in Lemma 4.5, we provide some preliminary lemmas which are similar to the correctness proofs in [Hutle and Widder 2005a].

Definition 4.1 *Instant t is a* clean broadcast time *for process p if (1) p executes* `line 17` *or* `line 6` *and thus sends $(p, ph, 1)$ to Λ at time t, and (2) for any ℓ, there are no (p, ph, ℓ) messages in transit at time t other than those sent by p at time t.*

Lemma 4.2 *If process p sends $(p, ph, 1)$ to Λ at some clean broadcast time t, then p does not send $(p, 1 - ph, 1)$ before $t + 2\Phi\tau^-$.*

Proof: Since t is a clean broadcast time, no messages $(p,\ ph,\ \ell)$ are in transit at t. Therefore, if p receives a $(p,\ ph,\ \ell)$ message for $1 \leq \ell \leq \Phi$ after time t, it is a correct response to one of p's $(p,\ ph,\ \ell)$ messages. The minimum time of such a round trip is $2\tau^-$.

By `line 10` and `line 17`, $(p, 1-ph, 1)$ is sent to Λ after Φ round trips. Thus, not before $t + 2\Phi\tau^-$. □

Lemma 4.3 *If correct process p sends $(p, ph, 1)$ to Λ at some time t, then p sends $(p, 1-ph, 1)$ to Λ by $t + 2\Phi\tau^+$.*

Proof: Process p sends $(p,\ 1-ph,\ 1)$ to Λ if it receives a (p, ph, Φ) message from some process in Λ and $phase_p$ is still ph. As $phase_p$ is only updated when p receives a (p, ph, Φ) message, it remains to show that p receives a (p, ph, Φ) message by time $t+2\Phi\tau^+$. A round-trip to a process in Λ requires at most time $2\tau^+$. By `line 10` and `line 17`, $(p, 1-ph, 1)$ is sent to Λ after Φ round trips. Thus p receives (p, ph, Φ) by time $t + 2\Phi\tau^+$. □

Lemma 4.4 *For Algorithm 1 with $\Phi > \Theta$ it holds that if process p sends $(p, ph, 1)$ to Λ at some clean broadcast time t, and there is a time t' such that t' is the first time after t at which p sends $(p, 1-ph, 1)$ to Λ, then t' is a clean broadcast time.*

Proof: By Lemma 4.2, process p does not send $(p, 1-ph, 1)$ to Λ before $t+2\tau^+$. All messages which are in transit to process p at time t are received by time $t + \tau^+$, of these all messages $(p, 1-ph, k)$ are ignored by process p (and hence no messages are sent). All messages $(p, 1-ph, k)$ which are in transit from process p to its neighbors are answered by them and these answers are received by p by $t + 2\tau^+$ and ignored as well since p is still in phase ph. Thus, no messages for phase $1-ph$ are in transit at time t' and thus the lemma holds. □

As initially there are no messages in transit, Lemma 4.4 shows that at any time some process p sends $(p, ph, 1)$ to Λ it does so at some clean broadcast time. Consequently, Lemma 4.2 and Lemma 4.3 provide bounds on the time between two such times. In the following, we use these properties to show that Algorithm 1 in fact implements a failure detector.

Lemma 4.5 *For Algorithm 1 with $\Phi > \Theta$ it holds that the set $suspects_p$ implements a perfect failure detector $\mathcal{P}_\Lambda$ with respect to the set of potential leaders Λ.*

Proof: We first show that no correct process $q \in \Lambda$ is ever suspected by some process $p \in \Lambda$. Assume by contradiction that some process p adds the correct process q to its suspect list $suspects_p$. This happens in `line 13`. By Lemma 4.2, at least $2\tau^+$ has passed since p has sent $(p, ph, 1)$. As q is added to the suspect list, p did not receive a response sent by q to this message.

This, however, is impossible due to the definition of Θ, the fact that $\Phi > \Theta$, the fact that q is not crashed, and `line 21`.

It remains to show that, when some process q does crash, each correct process p will eventually add it to $\mathit{suspects}_p$. At some time t after q crashes, p will start a new phase by sending $(p, ph, 1)$ to Λ and will thus send $(p, 1-ph, 1)$ eventually by Lemma 4.3. As these messages are sent via `line 17`, `line 13` is executed before. As q could not respond to p's $(p, 1-ph, 1)$ message and because t is a clean broadcast time (by Lemma 4.4) we have $\mathit{lastmsg}_p[q] = 0$ and q will be in the new $\mathit{suspects}_p$ set. □

Note that implementing $\mathcal{P}_\Lambda$ is not identical to implementing $\mathcal{P}_\Pi$, i.e., a perfect FD for all processes including the mute ones, since crashes by mute processes cannot be detected as they are externally non distinguishable. This is obviously different for implementing Ω: Since Ω only outputs one correct process, it is sufficient to choose the leader from a (large enough) subset of all processes. Therefore $\Omega_\Lambda = \Omega_\Pi$.

Lemma 4.6 *For Algorithm 1 with $\Phi > \Theta$ it holds that eventually all correct processes $p \in \Lambda$ have the same correct process $q \in \Lambda$ as their leader.*

Proof: Let t be the time the last process in Λ crashes during an execution and let process q be such that

$$q = \min_r \{r \mid r \in \Lambda \wedge r \text{ is correct}\}.$$

From `line 14` we see that the leader is selected out of the set of non suspected processes $r \in \Lambda$. By Lemma 4.5, all crashed processes in Λ will eventually be suspected, and therefore at some time after t all correct processes in Λ suspect the same processes, consequently all will determine the same minimum, i.e., the same leader, that is q. □

Lemma 4.7 *Algorithm 1 ensures that every process in Σ will choose $q \in \Lambda$ as its leader if q is the only process in Λ that keeps sending messages to Σ forever.*

Proof: Since there is only one process q that keeps sending (q) messages to Σ forever, eventually all messages (p) with $p \neq q$ are received. From then on, only (q) messages remain and by `line 24` processes in Σ will select q as leader whenever such message arrives. □

Theorem 4.8 *Algorithm 1 with $\Phi > \Theta$ implements Ω.*

Proof: By Lemma 4.6, eventually there is only one unique leader among Λ, let q denote this leader. It remains to show that the same processes becomes leader of the processes in Σ.

By `line 19` the leader $q \in \Lambda$ keeps sending (q) messages forever. Thus (by Lemma 4.7) q will also become the leader of the silent processes Σ. □

4.1 Discussions

Note that Lemma 4.7 shows that the leader of Σ emerges from the message pattern alone, and does not depend on the state of the processes. It can therefore be argued that the code for processes in Σ is self-stabilizing, while the code for Λ is not. To arrive at a self-stabilizing overall solution for message-driven leader election with silent processes, it would be sufficient to use adapt the algorithm for Λ. Indeed one could use any self-stabilizing Ω that fulfills the requisite in the lemma. In particular the algorithms of [Hutle and Widder 2005b; Hutle and Widder 2005a] can be adapted in this way by requiring every leader to broadcast its identifier, whenever it elects itself as leader.

Another interesting point is, that the dissemination of the leader to the processes in Σ is not restricted by the impossibility result of Hutle and Widder [2005a] in the same way as it is the case for the election of the leader in Λ. It follows that additional assumptions that are required to circumvent the impossibility result of Hutle and Widder [2005a] only have to consider processes in Λ respectively the links that connect them.

5 An Algorithm Based on a Global Time Base

In this section we assume that τ^+ and τ^- holds on all outgoing links of processes in Λ, that is within Λ and on the links connecting the silent processes Σ to the potential leaders Λ. The algorithm for processes $p \in \Lambda$ is again a variant of the bounded memory algorithm of Hutle and Widder [2005a] and it works similar to the algorithm of the previous section. The processes $p \in \Sigma$ work following an idea previously employed by Beauquier and Kekkonen-Moneta [1997][4] for their "type (1) failure detector": If some process has not received a message by p during a time when it has received Ψ messages by q it suspects p of being crashed. The difference, however, is that (a) our implementation is message-driven and (b) we do not *assume* some Ψ but *establish a value* for it from our synchrony assumptions.

In particular, active processes regularly send $(p, ph, 1)$ messages to mute processes. At mute processes, the value Ψ must be chosen such that it is ensured that no more than Ψ such $(p, ph, 1)$ messages sent by some correct process p are received between two receptions of $(q, ph, 1)$ messages sent by correct process q.

Unlike the technique used in Algorithm 1, the algorithm presented in this section does not only distribute the knowledge about the identity of the leader, but instead establishes a distributed time base that allows every process to time out crashed leader candidates. The ability to do so depends

[4] In fact the failure detector algorithm by Beauquier and Kekkonen-Moneta [1997] can quite easily be implemented in the partially synchronous model of Dwork et al. [1988].

Algorithm 2 FD Implementation based on a global time base

```
Code for processes $p \in \Lambda$:
$phase_p \in \{0,1\} \leftarrow 0$
$leader_p \in \Lambda \leftarrow \min_r\{r \in \Lambda\}$
$suspects_p \subset \Lambda \leftarrow \{\}$
$\forall q \in \Lambda:\ lastmsg_p[q] \in \{0,\dots,\Phi\} \leftarrow 0$

upon initialization do
    send $(p, phase_p, 1)$ to all

upon reception of $(p, ph, k)$ from $q$ do
    if $ph = phase_p$ and $k > lastmsg_p[q]$ then
        $lastmsg_p[q] \leftarrow k$
        if $k < \Phi$ then
            send $(p, phase_p, k+1)$ to $q$
        else
            $suspects_p \leftarrow \{r \mid r \in \Lambda \wedge lastmsg_p[r] = 0\}$
            $leader_p \leftarrow \min_r\{r \mid r \in \Lambda - suspects_p\}$
            $phase_p \leftarrow 1 - phase_p$
            $\forall r \in \Lambda:\ lastmsg_p[r] \leftarrow 0$
            send $(p, phase_p, 1)$ to all

upon reception of $(q, ph, k)$ from $q$ do
    send $(q, ph, k)$ to $q$

Code for processes $p \in \Sigma$:
$leader_p \in \Lambda \leftarrow \min_r\{r \in \Lambda\}$
$suspects_p \in \Lambda \leftarrow \{\}$
$\forall r, s \in \Lambda : count_p[r,s] \leftarrow 0$        // messages from $r$ since last message from $s$

upon reception of some message from $q \in \Lambda$ do
    $suspects_p = suspects_p - \{q\}$
    for all $g \in \Lambda - \{q\}$ do
        if $g \notin suspects_p$ then
            $count_p[q,g] \leftarrow count_p[q,g] + 1$
            if $count_p[q,g] > \Psi$ then
                $suspects_p = suspects_p \cup \{g\}$
        $count_p[g,q] \leftarrow 0$
    $leader_p \leftarrow \min_r\{r \mid r \in \Lambda - suspects_p\}$
```

heavily on the structure of the established time base which — in message-driven systems — depends on the specific algorithm and thus its message pattern. In Algorithm 2, the value Ψ used in the code of mute processes represents the "knowledge" of the mute processes on the inner details (message pattern) of the active processes. Consequently this technique for mute processes is not a generic method as is the technique of the previous section.

The following Lemmas 5.1 and 5.2 follow the same argument as their counterparts for Algorithm 1, we therefore leave the details of the proofs to the reader.

Lemma 5.1 *For Algorithm 2 with $\Phi > \Theta$ it holds that the set $suspects_p$ implements a perfect failure detector $\mathcal{P}_\Lambda$ with respect to the set of potential* *leaders* Λ.

Lemma 5.2 *For Algorithm 2 with* $\Phi > \Theta$*, it holds that eventually all correct processes in* Λ *have the same correct process* $q \in \Lambda$ *as their leader.*

Lemma 5.2 shows that all processes in Λ will eventually agree on an correct process as their leader. Before we can show that all processes in Σ will agree with them, we examine the minimum and maximum times that pass at silent processes between receiving messages from potential leaders, to arrive at a value for Ψ. The intuition behind Ψ is that it is possible to bound the ratio between the number of messages a process $q \in \Sigma$ receives from two potential leaders. To do this we provide the following two lemmas that can be proven like Lemma 4.2 and Lemma 4.3.

Lemma 5.3 *For Algorithm 2, the minimum time between process* $p \in \Lambda$ *sends two subsequent* $(p, -, 1)$ *messages to all is* $D_{min} = 2\Phi\tau^-$.

Lemma 5.4 *For Algorithm 2, the maximum time between correct process* $p \in \Lambda$ *sends two subsequent* $(p, -, 1)$ *messages to all is* $D_{max} = 2\Phi\tau^+$.

Lemma 5.5 *For Algorithm 2, the minimum time between process* $q \in \Sigma$ *receives two* $(p, -, 1)$ *messages from correct process* $p \in \Lambda$ *is* $\max\{D_{min} - \varepsilon, 0\}$.

Proof: By Lemma 5.3, the minimum time between p sending two $(p, -, 1)$ messages to all processes is D_{min}. Let p send one such message at time t and the next one at time $t' = t + D_{min}$. The message sent at time t is received by q in the interval $[t + \tau^-, t + \tau^+]$ and correspondingly the message sent at time t' is received in the interval $[t' + \tau^-, t' + \tau^+]$. The shortest possible interval between the two receptions therefore is $t' + \tau^- - (t + \tau^+) = (t - t') - \tau^+ + \tau^- = D_{min} - \varepsilon$ if this value is greater than 0. Otherwise the the message sent at time t' is received by q before the message sent at time t. □

Lemma 5.6 *For Algorithm 2 with* $\Phi > \Theta$*, the minimum time between process* q *receiving two* $(p, -, 1)$ *messages from correct process* $p \in \Lambda$ *is* $D_{min} - \varepsilon > 0$.

Proof: We show that if we choose the value $\Phi > \Theta$, the term of Lemma 5.5, $\max\{D_{min} - \varepsilon, 0\} = D_{min} - \varepsilon$. That is what has to be shown is that $D_{min} - \varepsilon > 0$. From Lemma 5.3 and the assumption that $\Phi > \Theta$, it follows that $D_{min} = 2\Phi\tau^- > 2\Theta\tau^- = 2\tau^+$, such that we obtain $D_{min} - \varepsilon > 2\tau^+ - \varepsilon > 0$. □

Lemma 5.7 *For Algorithm 2, the maximum time between process* q *receives two* $(p, -, 1)$ *messages from process* $p \in \Lambda$ *is* $D_{max} + \varepsilon$.

Proof: This proof proceeds in a way similar to the proof of Lemma 5.5. By Lemma 5.4, the maximum time between p sending two $(p, -, 1)$ messages to all processes is D_{max}. Let p send one such message at time t and the next one at time $t' = t + D_{max}$. The message sent at time t is received by q in the interval $[t + \tau^-, t + \tau^+]$ and correspondingly the message sent at time t' is received in the interval $[t'+\tau^-, t'+\tau^+]$. The longest possible interval between two receptions is thus $t' + \tau^+ - (t + \tau^-) = (t - t') + \tau^+ - \tau^- = D_{max} + \varepsilon$. □

In Lemma 5.8 we finally obtain the value for Ψ which the silent processes can use to timeout active processes.

Lemma 5.8 *For the Algorithm 2 with* $\Phi > \Theta$ *and* $\Psi > \Theta + 1$*, it holds that eventually all correct processes* $p \in \Sigma$ *have* q *from Lemma 5.2 as leader.*

Proof: From `line 31` we see that the leader is selected out of the set of non suspected processes $r \in \Lambda$. In order to show that eventually a unique leader is chosen, we have to show that eventually all crashed processes in Λ are suspected forever and that eventually no correct processes is suspected by any process. (On the one hand, these properties correspond the strong completeness and eventual strong accuracy [Chandra and Toueg 1996] where only crashes and correctness of processes $\in \Lambda$ are detected. On the other hand, these properties correspond to those of $\mathcal{P}_\Lambda$, with requirements on the *suspect* sets of all processes in Π.) Therefore, the processes in Σ will eventually suspect the same processes as those in Λ and therefore determine the same minimum id, that is the same leader, i.e., q from Lemma 5.2.

We first show that eventually no correct process in Λ is suspected by any $p \in \Sigma$. Some process o becomes suspected by `line 29` if for some other process r, $count_p[r, o] > \Psi$. We thus have to show that for all correct processes $o \in \Lambda$ and all processes $r \in \Lambda$, eventually (more precisely, after the initial step o) it holds true forever that $count_p[r, o] \leq \Theta + 1 < \Psi$. Assume that some process $p \in \Sigma$ receives a message from o at time t such that $count_p[r, o] = 0$ then. Note that by `line 24` o is not suspected at time t by p.

At process p, $count_p[r, o]$ increases every time p receives a message from r until either $count_p[r, o] > \Psi$, i.e., until o is suspected by `line 29` or otherwise p receives a message by o and $count_p[r, o]$ is set to 0 again. We show that, after t, the latter must always happen if o is correct.

According to Lemma 5.7, the next message by o is received by p by time $t' = t + D_{max} + \varepsilon$. We have to show that p cannot receive more than Ψ messages from r by t', i.e., in any interval of size $D_{max} + \varepsilon$. By Lemma 5.6, p can receive messages at most every $D_{min} - \varepsilon$ time units. Since the first of such messages can be received arbitrarily close to t, at most

$$\Psi = 1 + \frac{D_{max} + \varepsilon}{D_{min} - \varepsilon} = 1 + \frac{2\Phi\tau^+ + \varepsilon}{2\Phi\tau^- - \varepsilon}$$

messages can be received by time t'. We obtain

$$(\Psi - 1) \cdot (2\Phi\tau^- - \varepsilon) = 2\Phi\tau^+ + \varepsilon$$

such that we find

$$\Psi \cdot (2\Phi\tau^- - \varepsilon) = 2\Phi \cdot (\tau^+ + \tau^-).$$

We arrive at

$$\Psi = \frac{2\Phi(\tau^+ + \tau^-)}{2\Phi\tau^- - \varepsilon} \geq \frac{2\Phi(\tau^+ + \tau^-)}{2\Phi\tau^-} = \Theta + 1$$

which concludes the first part of the proof.

For showing that eventually every crashed process $o \in \Lambda$ is suspected by $p \in \Sigma$ it suffices to see that there are at least 2 correct processes in Λ which keep sending messages for ever. Let q be such a correct process. Since o crashes, there must be a last message sent from o received by p. When this message is received, p does not suspect o by `line 24`. After that there must be messages sent by q which will be received by p such that by `line 27` $count_p[q, o]$ increases until the condition of `line 29` evaluates to true and o becomes suspected. Since no message of o will be received by p after that, `line 24` will never be executed for o such that o remains suspected forever. $\square$

From Lemma 5.2 and Lemma 5.8 it follows directly, that

Theorem 5.9 *Algorithm 2 with* $\Phi > \Theta$ *and* $\Psi > \Theta + 1$ *implements* Ω.

6 The Structure of Message-driven Algorithms in Comparison

Considering how the message-driven implementations of failure detectors in [Hutle and Widder 2005a] work, our lower bound results appear to be obvious: Failure detection is done by comparing round-trips with at least $f + 1$ other processes. This ensures that there will always be timely (bounded by lower and upper bounds) communication between at least 2 correct processes. This communication establishes some kind of time-base — or a source of synchrony — which is local between these two processes. It seems that this is necessary and this intuition is confirmed in Theorem 3.1. Regarding the structure, the algorithms in [Hutle and Widder 2005a] are geocentric. Each process does failure detection with its neighbors via some sort of round-trip measurement. This local time-base allows to solve certain problems (like timing out a crashed process) within these two processes.

Our first algorithm from Section 4 shares the same geocentric structure inside the set Λ. Eventually only the process elected leader inside Λ communicates (asynchronously) with the remaining processes. Consequently, the processes in Σ are passive recipients of the outcome of an election process.

Our Algorithm 2 in Section 5 can be seen as a competition. Processes inside Λ compete in convincing processes in Σ that they are still alive. The election process is thus not taken inside Λ and then just the result is sent to Σ, but the election is done on a global basis. So somehow, our second algorithm can be seen as intermediate step between Algorithm 1 and the algorithms in [Hutle and Widder 2005a]. The election is done in the whole system, while the set of leader candidates is restricted to Λ. The major difference between Algorithm 2 and [Hutle and Widder 2005a] is the one-way communication that is incoming to the processes in Σ.

The structure of the message-driven failure detector implementations given in e.g. [Le Lann and Schmid 2003; Widder et al. 2005; Hermant and Widder 2005] is fundamentally different. They are based on the message-driven variant of the consistent broadcasting primitive by Srikanth and Toueg [1987]. These algorithms are built upon synchronized integer clocks where the precision of the clocks (the maximum deviation between any two clock values of correct processes) as well as the accuracy (the rates at which the clock values increase) can be used to establish a dimensionless timeout value which allows to detect failures.

We believe that our results contribute to the general understanding of the term "synchrony".[5] What are sources of synchrony? How many do we need to solve certain problems in a fault-tolerant manner and how strong do they have to be? We believe that due to the interleaving of event generation with synchrony assumptions, the basic properties of synchrony are not perfectly understood by now. By investigating different models and observing the commonalities, we hope that it is possible to eventually get an abstract notion of synchrony (or time) in fault-tolerant distributed computations.

The timing assumption on which our Ω implementations are based on are rather conservative in order not to distract from the event generation issue. Similar to [Dwork et al. 1988; Chandra and Toueg 1996] one might assume that Θ is unknown, i.e., in every execution there may be a different Θ. Using such assumptions, the values Φ and Ψ would have to be adaptive, i.e., they must be increased either with every phase or, for instance, whenever a process was wrongly suspected of being crashed. Note carefully that in this case the proof of correctness would become more involved as it could be ensured only from some point in time on that no (p, ph, k) messages are in transit, when process p starts a new phase with phase number ph; a property

[5]Note that in the context of fault-tolerant distributed computing, at the latest with the results of Dolev et al. [1987], the term synchrony deviates heavily from the original meaning of simultaneity, i.e., events happening at the same time [Einstein 1905].

which is crucial for the accuracy of the failure detector.

7 Conclusions

In this paper we explored the required properties for implementing the eventual leader oracle Ω in the context of message-driven algorithms. For this, we found limits for algorithms that implement Ω under the given system requirements: We showed that it is harder to implement the failure detector Ω in message-driven systems than it is in time-driven systems by proving that strictly more processes are required to tolerate a given number of faults. The analysis reveals that the absence of synchrony or timing assumptions regarding self-receptions is central for our results. It is quite obvious that an assumption like some lower bound on self-receptions would allow to implement a simulation for partially synchronous models like the FAR model [Fetzer et al. 2005]. Note also that using self-receptions for timing out events also dismisses the system requirement for doing computations only when necessary as we again get periodic task activations independently of the current state of the system.

Previous results [Hutle and Widder 2005a] showed that message-driven semantics are weaker than time-driven semantics with respect to self-stabilization. Here, we have shown that message-driven semantics are weaker in non self-stabilizing systems as well. Apart from resilience, to implement Ω other assumptions have to be stronger as well: In order to guarantee liveness, message-driven solutions require reliable links or bounded message-loss, such that enough messages always remain to trigger computational steps. In contrast, time-driven solutions typically only demand the eventual absence of message-loss to allow accurate discrimination between crashed and alive processes; see e.g. [Chandra and Toueg 1996; Aguilera et al. 2003; Aguilera et al. 2004]. Moreover, $f+2$ active processes are required for message-driven systems, whereas in synchronous time-driven systems f active processes suffice.

Acknowledgements. The authors are grateful to Martin Hutle for many enlightening discussions.

References

Aguilera, M. K., Delporte-Gallet, C., Fauconnier, H., and Toueg, S. 2003. On implementing Omega with weak reliability and synchrony assumptions. In *Proceeding of the 22nd Annual ACM Symposium on Principles of Distributed Computing (PODC'03)*. ACM Press, New York, NY, USA, 306–314.

Aguilera, M. K., Delporte-Gallet, C., Fauconnier, H., and Toueg, S. 2004. Communication-efficient leader election and consensus with limited link synchrony. In *Proceedings of the 23th ACM Symposium on Principles of Distributed Computing (PODC'04)*. ACM Press, St. John's, Newfoundland, Canada, 328–337.

Beauquier, J. and Kekkonen-Moneta, S. 1997. Fault-tolerance and self-stabilization: Impossibility results and solutions using self-stabilizing failure detectors. *International Journal of Systems Science 28,* 11, 1177–1187.

Chandra, T. D., Hadzilacos, V., and Toueg, S. 1996. The weakest failure detector for solving consensus. *Journal of the ACM 43,* 4 (June), 685–722.

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

Dolev, D., Dwork, C., and Stockmeyer, L. 1987. On the minimal synchronism needed for distributed consensus. *Journal of the ACM 34,* 1 (Jan.), 77–97.

Dwork, C., Lynch, N., and Stockmeyer, L. 1988. Consensus in the presence of partial synchrony. *Journal of the ACM 35,* 2 (Apr.), 288–323.

Ebergen, J. C. 1991. A formal approach to designing delay-insensitive circuits. *Distributed Computing 5*, 107–119.

Einstein, A. 1905. Zur Elektrodynamik bewegter Körper. *Annalen der Physik 322,* 10, 891–921.

Fetzer, C., Schmid, U., and Süsskraut, M. 2005. On the possibility of consensus in asynchronous systems with finite average response times. In *Proceedings of the 25th International Conference on Distributed Computing Systems (ICDCS'05).* IEEE Computer Society, Washington, DC, USA, 271–280.

Fischer, M. and Lamport, L. 1982. Byzantine generals and transaction commit protocols. Technical Report 62, SRI International.

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

Fuegger, M., Schmid, U., Fuchs, G., and Kempf, G. 2006. Fault-Tolerant Distributed Clock Generation in VLSI Systems-on-Chip. In *Proceedings of the Sixth European Dependable Computing Conference (EDCC-6).* IEEE Computer Society Press, 87–96.

Gärtner, F. C. and Pleisch, S. 2001. (Im)possibilities of predicate detection in crash-affected systems using interrupt-style failure detectors. In *Brief Announcements — 15th International Symposium on DIStributed Computing (DISC 2001)*, J. Welch, Ed. Technical Report TR-01-7. Departamento de Informática, Faculdade de Ciências da Universidade de Lisboa, Lisboa, Portugal, 7–12. http://www.di.fc.ul.pt/publications/di-fcul-tr-01-7_document.pdf.

Guerraoui, R. and Schiper, A. 1996. "Γ-accurate" failure detectors. In *Proceedings of the 10th International Workshop on Distributed Algorithms (WDAG'96)*, Ö. Babaoğlu, Ed. LNCS, vol. 1151. Springer Verlag, Berlin / Heidelberg, 269–286.

Hermant, J.-F. and Widder, J. 2005. Implementing reliable distributed real-time systems with the Θ-model. In *Proceedings of the 9th International Conference on Principles of Distributed Systems (OPODIS 2005).* LNCS, vol. 3974. Springer Verlag, Pisa, Italy, 334–350.

Hutle, M., Malkhi, D., Schmid, U., and Zhou, L. 2006. Brief announcement: Chasing the weakest system model for implementing omega and consensus. In *Proceedings Eighth International Symposium on Stabilization, Safety, and Security of Distributed Systems (formerly Symposium on Self-stabilizing Systems) (SSS 2006).* LNCS. Springer Verlag, Dallas, USA, 576–577.

Hutle, M., Malkhi, D., Schmid, U., and Zhou, L. 2008. Chasing the weakest system model for implementing omega and consensus. *IEEE Transactions on Dependable and Secure Computing*. (to appear).

Hutle, M. and Widder, J. 2005a. On the possibility and the impossibility of message-driven self-stabilizing failure detection. In *Proceedings of the Seventh International Symposium on Self Stabilizing Systems (SSS 2005).* LNCS, vol. 3764. Springer Verlag, Barcelona, Spain, 153–170. Appeared also as brief announcement in *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing (PODC'05)*.

Hutle, M. and Widder, J. 2005b. Self-stabilizing failure detector algorithms. In *Proc. IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN'05)*. IASTED/ACTA Press, Innsbruck, Austria, 485–490.

Lamport, L. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21,* 7, 558–565.

Le Lann, G. and Schmid, U. 2003. How to implement a timer-free perfect failure detector in partially synchronous systems. Tech. Rep. 183/1-127, Department of Automation, Technische Universität Wien. January. (Replaced by Research Report 28/2005, Institut für Technische Informatik, TU Wien, 2005.).

Lynch, N. 1996. *Distributed Algorithms.* Morgan Kaufman Publishers, Inc., San Francisco, USA.

Malkhi, D., Oprea, F., and Zhou, L. 2005. Ω meets paxos: Leader election and stability without eventual timely links. In *Proceedings of the 19th Symposium on Distributed Computing (DISC'05)*. LNCS, vol. 3724. Springer Verlag, Cracow, Poland, 199–213.

Pease, M., Shostak, R., and Lamport, L. 1980. Reaching agreement in the presence of faults. *Journal of the ACM 27,* 2 (April), 228–234.

Robinson, P. and Schmid, U. 2008. Brief announcement: The asynchronous bounded-cycle model. In *Proceedings of the 27th ACM Symposium on Principles of Distributed Computing (PODC'08)*. ACM Press, 423. (extended version to appear at SSS'08).

Santoro, N. and Widmayer, P. 1989. Time is not a healer. In *Proc. 6th Annual Symposium on Theor. Aspects of Computer Science (STACS'89)*. LNCS 349. Springer-Verlag, Paderborn, Germany, 304–313.

Srikanth, T. K. and Toueg, S. 1987. Optimal clock synchronization. *Journal of the ACM 34,* 3 (July), 626–645.

Widder, J. 2003. Booting clock synchronization in partially synchronous systems. In *Proceedings of the 17th International Symposium on Distributed Computing (DISC'03)*. LNCS, vol. 2848. Springer Verlag, Sorrento, Italy, 121–135.

Widder, J. 2004. Distributed computing in the presence of bounded asynchrony. Ph.D. thesis, Vienna University of Technology, Fakultät für Informatik.

Widder, J., Le Lann, G., and Schmid, U. 2005. Failure detection with booting in partially synchronous systems. In *Proceedings of the 5th European Dependable Computing Conference (EDCC-5)*. LNCS, vol. 3463. Springer Verlag, Budapest, Hungary, 20–37.