Optimal Message-Driven Implementation of Omega with Mute Processes

Martin Biely       Josef Widder

November 12, 2008

Abstract

We consider the complexity of algorithms in message-driven models, i.e., models of distributed computations where events can only be caused by message receptions but not by the passage of time. Hutle and Widder (2005) have shown that there is no self-stabilizing implementation of the eventually strong failure detector, and thus the eventual leader oracle Ω in such models under certain assumptions. Under stronger assumptions it was shown that even the eventually perfect failure detector can be implemented in 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. After that, we provide an efficient message-driven implementation of Ω. Our algorithm is efficient in the sense that not all processes have to send messages forever, which is an improvement to previous message-driven failure detector implementations.
1 Introduction

Fault-tolerant agreement problems are crucial for both practical applications of distributed algorithms, as well as for understanding the principles of distributed computations. In this context, specifically the role of time—or rather synchrony—is heavily researched, for example with respect to consensus, the problem of agreeing on a common value despite faults [1, 2, 3, 4]. As a result of research on consensus, it was shown that if just crash faults are contemplated, synchrony can be encapsulated by failure detectors [4] and that the eventual leader oracle Ω is the weakest failure detector (FD) to allow solving consensus [5]. 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 on its own that can be solved with synchrony assumptions, and much work focused on contemplating timing models to that end and in fact very weak models have been established [6, 7, 8]. But timing is not the sole parameter of distributed computing models: others are for example atomicity of events (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 time-driven models, events can be triggered locally by the passage of time, i.e., by clocks or timers. In the message-driven model of execution, events 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. Algorithms that perform steps only upon message receptions were discussed in [9] and named “asynchronous” there (similar to asynchronous network protocols that operate by explicit handshaking).

Although 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 [10]) circuits. Asynchronous hardware differs from synchronous one in the way computations are triggered. In synchronous designs, there exists a central clock whose ticks trigger each component (flip-flop) conceptually simultaneously. The inputs of a component are ready when the tick is generated, and its outputs are present at the next stage of the logic at the beginning of the next tick. The concept of periodically executed steps is not appropriate for asynchronous chip design (due to the lack of a central clock). In [11] it
was shown 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.

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. Only little work exists in the context of consensus that considers how events in a distributed computation are triggered [12]. In [13] it was shown that there is a difference between the expressiveness of time-driven and message-driven execution models, by showing that \( \Omega \) cannot be implemented with a self-stabilizing message-driven protocol under certain assumptions where self-stabilizing time-driven protocols [14] can solve the problem. By strengthening the assumptions it was shown that the problem has message-driven implementations as well. The question of 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 was not analyzed. With the present paper, we intend to contribute to closing this gap: The aforementioned message-driven implementations of \( \Omega \) require a system 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 \( \Omega \) are known that require just \( f + 1 \) processes, therefore the question arises whether the algorithms in [13] are optimal regarding the required number of processes. In this paper we answer this question in the affirmative. We present an \( f + 2 \) lower bound on the number of processes required for message-driven implementations of \( \Omega \).

Additionally, we consider efficiency issues: It is known that for time-driven systems, communication efficient failure detector implementations are possible [15]. Here, communication efficiency refers to the number of processes that have to keep sending messages forever. All known message-driven failure detector implementations [13, 16, 17, 18] require all correct processes to keep sending messages forever. We show that this is not necessary for implementing \( \Omega \). Our algorithm requires 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 algorithm is optimal in this respect.

This reduced complexity may not seem particularly relevant at first sight, but the remarkable fact is that for solving consensus with \( \Omega \) 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 \( n = 3 \) and \( f = 1 \) all processes have to send messages. In all other systems in
which consensus can be solved, our implementation of $\Omega$ allows $n - f - 2 > \frac{n}{2} - 2 \geq 0$ processes to remain silent.

1.1 Contribution

To the best of our knowledge, this paper establishes for the first time 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 $\Omega$ with $f + 1$ processes even under far less restrictive synchrony assumptions [15].

We also present an algorithm that shows that this bound is tight. Additionally, the algorithm allows some processes to remain silent throughout the execution. We thus contribute to the comparison of time-driven and message-driven protocols.

Knowing message-driven implementations of failure detectors [13], our 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 — for these two processes that allows to solve certain problems (like timing out crashed processes).

We believe that our results contributes to the general understanding of the term “synchrony”. What are sources of synchrony? How many do we need to solve certain problems in a fault-tolerant manner? What semantics do these sources have? 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.

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 $\Omega$ that is optimal with respect to the derived resilience bound.
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 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 \( \Pi = \{p | 1 \leq p \leq N\} \).

The processes perform a distributed computation which proceeds by the computational steps of processes in \( \Pi \). 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.

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 \( \Pi \) may fail during an execution. Since we are interested in FD implementations where only some processes ever send messages, we introduce \( \Lambda = \{p | 1 \leq p \leq n\} \), which is the set of \( n \leq N \) active processes, i.e., processes that send messages. The set of silent processes will be called \( \Sigma = \Pi - \Lambda \). We define \( s = |\Sigma| \geq 0 \) and since \( N = |\Pi| \), we have \( N = n + s \).

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 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 for 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 \( q \neq p \). For simplicity 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. Our algorithm does not know these values. Indeed the knowledge of the values would be useless as processes cannot measure time. Instead we assume the knowledge of the ratio \( \Theta = \tau^+/\tau^- \). In our analysis we also use the transmission uncertainty \( \varepsilon = \tau^+ - \tau^- \). It has been shown in [19] 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 must be bounded by some \( \Theta \). In other words, \( \tau^+ \) and \( \tau^- \) may change during the execution, as long as their ratio continues to be bounded by \( \Theta \). Note that \( \Theta \) is in fact the only (time-related) value that processes can observe.

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 Events

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 \textit{self reception step}. If at least one message in \( M \) was sent by some process \( q \neq p \) then the step is called \textit{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 \textit{potentially final extrinsic reception step}. By \textit{potentially final non self reception step} we denote all steps that are either a potentially final extrinsic reception step or an initial step. (A potentially final extrinsic reception step occurs at \( p \) when \( p \) has received all messages that causally precede 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, both of which will only output failure information about processes in \( \Lambda \). The failure detector \( \Omega \) [5] outputs a single process (its leader estimate), which eventually must be the same correct process at all processes. The formal definition for \( \Omega \) reads as follows.
**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 $P$. It was defined [4] 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 $\Lambda$. Therefore we define the following generalization of the perfect FD. Thus we define $P_\Lambda$ via two following 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 introduced $\Gamma$-accurate FDs [20] which are similar to $P_\Lambda$. $P_\Lambda$, however, restricts both accuracy and completeness to some fixed subset $\Lambda$ of all processes while $\Gamma$-accurate FDs only restrict accuracy properties to some fixed subset $\Gamma$.

We do not restrict the semantics of the algorithms that use our FDs, i.e., classic query based execution models [4] can be employed as well as interrupt based models; discussions on the respective expressiveness can be found in [12]. 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 $\Omega$ — changes.

### 3 Lower Bound on the Number of Processes

We show that it is impossible to implement $\Omega$ 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 $I$ of $\Omega$. In the following, we show how $I$ must behave if $n - 1$ processes in $\Lambda$ crash during an execution. Then we consider executions where just $n - 2$ processes in $\Lambda$ crash. By indistinguishability to the first execution, $I$ violates the properties of $\Omega$ thus providing the required contradiction.
Theorem 3.1 (Lower Bound)  There is no correct message-driven implementation of \( \Omega \) in our model if \( n \leq f + 1 \).

Proof: Assume by contradiction that there exists a message-driven implementation \( I \) of \( \Omega \), for a system where \( n \leq f + 1 \).

Let \( E_1 \) be the set of all executions of \( I \) where \( n - 1 \) processes in \( \Lambda \) 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 \( \text{leader}_p = p \) or \( \text{leader}_p = r \in \Sigma \) permanently—in both cases the leader estimate of \( p \) satisfies (EL).

Let \( E_2 \) be the set of all executions of \( I \) where \( n - 2 \) processes in \( \Lambda \) crash initially and there are two correct processes \( p, q \in \Lambda, p \neq q \). Note that just \( n - 2 < f \) processes in \( \Lambda \) are faulty such that there can be faulty processes also in \( \Sigma \) in \( E_2 \). Let all executions in \( E_2 \) 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 \( E_2 \). Let these finite executions \( E'_2 \) have some potentially final non self reception step \( s \) at some process \( p \) as their final step. For \( p \), every execution \( e \in E'_2 \) is indistinguishable from some finite prefix execution \( e_1 \in E_1 \) 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, 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 \( \text{leader}_p = p \) or \( \text{leader}_p = r \in \Sigma \) for some \( \ell \). This constructive argument can be applied to every potentially final non self reception step at any of the two correct processes in \( \Lambda \), 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 \( \text{leader}_v = v \) or \( \text{leader}_v = r \in \Sigma \) upon every (following) potentially final extrinsic reception step. It follows that they cannot set \( \text{leader}_p = p \) and \( \text{leader}_q = q \) permanently, as this would violate the “the same correct process” requirement. Thus, \( v \in \{p, q\} \) must set \( \text{leader}_v = r \in \Sigma \). 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 \( \Lambda \) crash in all \( E_2 \) executions. That is, at least one process in \( \Lambda \) can crash in such executions. Since \( r \) is in \( \Sigma \), 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 \( \text{leader}_p = r \) but \( r \) is crashed which violates (EL). We again reach a contradiction.

**Corollary 3.2** There is no correct message-driven implementation of \( \Omega \) in our model if \( N \leq f + 1 \).

**Corollary 3.3** There is no correct message-driven implementation of \( \Omega \) if \( N \leq f + 1 \) where processes never send messages to themselves.

Corollary 3.2 shows that the self-stabilizing algorithms in [13] are optimal regarding the number of processes required. (The impossibility of [13], however, even holds if there are synchrony assumptions on self-receptions.) 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). Since the algorithms of this paper do not create self receptions, Corollary 3.3 shows that our algorithms are optimal as well.

### 4 A Matching Algorithm

The algorithm has different code for the processes of \( \Lambda \) and \( \Sigma \). The code for processes \( p \in \Lambda \) is a variant of the bounded memory algorithm of [13]. Each active process \( p \) exchanges \((p, ph, k)\) messages with the other processes in \( \Lambda \), where \( ph \) is the phase number 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 all \( q \in \Lambda \), \( p \) holds a variable \( \text{lastmsg}_p[q] \), where it stores the highest integer \( k \) received in a \((p, ph, k)\) reply from \( q \). If \( \Phi \) is chosen properly, \( p \) can correctly suspect a process \( q \) of being crashed upon termination of \( \Phi \) round trips if there was no round trip terminated by \( q \).

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 \( \Lambda \) and \( \Sigma \) also obey the \( \Theta \) assumption (cf. Section 2.1).

**Lemma 4.1** For Algorithm 1 with \( \Phi > \Theta \) it holds that the set \( \text{suspects}_p \) implements a perfect failure detector \( \mathcal{P}_\Lambda \) with respect to the set of potential leaders \( \Lambda \).

**Proof:** To show that \( \text{suspects}_p \) acts as a perfect failure detector for processes in \( \Lambda \), we have to show (LSC) and (LSA).
Algorithm 1 Failure Detector Implementation

Code for processes $p \in \Lambda$:

1: $\text{phase}_p \in \{0, 1\} \leftarrow 0$
2: $\text{leader}_p \in \Lambda \leftarrow \min_r \{r \in \Lambda\}$
3: $\text{suspects}_p \subset \Lambda \leftarrow \{\}$
4: $\forall q \in \Lambda : \text{lastmsg}_p[q] \in \{0, \ldots, \Phi\} \leftarrow 0$
5: upon initialization do
6: send $(p, \text{phase}_p, 1)$ to $\Lambda$

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

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

Code for processes $p \in \Sigma$:

22: $\text{leader}_p \in \Lambda \leftarrow \min_r \{r \in \Lambda\}$
23: upon reception of $(q)$ from some $q \in \Lambda$ do
24: $\text{leader}_p \leftarrow q$

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 $\text{suspects}_p$. Then $p$ must have performed $\Phi$ round-trips (via line 11 and line 7) since the beginning of the current phase with some other process, while not receiving a response from $q$ to the message $p$ has sent in line 6 or line 17. This, however, is impossible due to the definition of $\Theta$ in Section 2.1, the fact that $\Phi > \Theta$ and the fact that $q$ is not crashed.

It remains to show that, when some process $q$ does crash, $p$ will eventually add it to $\text{suspects}_p$. At some time after $q$ crashes, $p$ will start a new phase, and subsequently perform $\Phi$ round-trips with some other correct process $r$, after which line 13 will be executed for a phase during which $q$ remained silent, therefore $\text{lastmsg}_p[q] = 0$ and $q$ will be in the new $\text{suspects}_p$ 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. 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 ΩΛ = ΩΠ.

Lemma 4.2 For Algorithm 1 with Φ > Θ it holds that eventually all correct processes p ∈ Λ have the same correct process q ∈ Λ 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
\[ q = \min_r \{ r \mid r \in \Lambda \land r \text{ is correct} \}. \]

From line 14 we see that the leader is selected out of the set of non suspected processes r ∈ Λ. By Lemma 4.1, 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 4.3 Algorithm 1 ensures that processes in Σ will choose the process q ∈ Λ as 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 ≠ 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 4.4 Algorithm 1 with Φ > Θ implements the eventual leader oracle Ω.

Proof: By Lemma 4.2, 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 ∈ Λ keeps sending (q) messages forever. Thus (by Lemma 4.3) q will also become the leader of the silent processes Σ.

5 Discussions

Note that Lemma 4.3 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 adapt the algorithm for Λ. Indeed one could use any self-stabilizing implementation of Ω that fulfills the requisite in the lemma. In particular,
the algorithms of [13, 21] 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 [13] 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 [13] only have to consider processes in Λ respectively the links that connect them.

6 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 [22].

Previous results [13] 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. [4, 6, 15].

References


