Partial Synchrony Based on Set Timeliness

[Extended Abstract]

Marcos K. Aguilera
Microsoft Research Silicon Valley
Mountain View, CA, USA

Carole Delporte-Gallet*
Université Paris 7
Paris, France

Hugues Fauconnier*
Université Paris 7
Paris, France

Sam Toueg†
University of Toronto
Toronto, ON, Canada

ABSTRACT

We introduce a new model of partial synchrony for read-write shared memory systems. This model is based on the notion of *set timeliness*—a natural and straightforward generalization of the seminal concept of *timeliness* in the partially synchrony model of Dwork, Lynch and Stockmeyer [8].

Despite its simplicity, the concept of set timeliness is powerful enough to describe the first partially synchronous system for read/write shared memory that separates *consensus* and *set agreement*: we show that this system has enough timeliness for solving set agreement but not enough for solving consensus.

Set timeliness also allows us to define a family of partially synchronous systems of n processes, denoted $\mathcal{S}_n^k$ ($1 \leq k \leq n-1$), which closely matches the family of k-anti-Ω failure detectors that were recently shown to be the weakest failure detectors for the k-*set agreement* problem: We prove that for $1 \leq k \leq n-1$, $\mathcal{S}_n^k$ is synchronous enough to implement k-anti-Ω but not enough to implement $(k-1)$-anti-Ω.

The results above show that set timeliness can be used to study and compare the partial synchrony requirements of problems that are strictly weaker than consensus.

Categories and Subject Descriptors

B.3.2 [**Memory structures**]: Design Styles—*Shared memory*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*; D.4.1 [**Operating systems**]: Process Management—*Concurrency*; F.1.1 [**Computation by Abstract Devices**]: Models of Computation; F.2.m [**Analysis of Algorithms and Problem Complexity**]: Miscellaneous

General Terms

Algorithms, Design, Theory

*This author was supported by grant ANR-08-VERSO-SHAMAN.

†This author was partially supported by the National Science and Engineering Research Council of Canada.

PODC'09, August 10–12, 2009, Calgary, Alberta, Canada.

Keywords

Set agreement, consensus, partial synchrony, timeliness, algorithms, failure detectors, impossibility, bivalency

1. INTRODUCTION

The concept of partial synchrony, introduced in the seminal work of Dwork, Lynch and Stockmeyer [8], is based on the notion of *timeliness*, e.g., an upper bound Φ on relative process speeds: "in any contiguous interval containing Φ real-time steps, every correct process must take at least one step. This implies no correct process can run more than Φ times slower than another." In the partially synchronous systems in [8], *all* the processes are (eventually) timely relative to each other.

To define partially synchronous systems that are weaker than those in [8], but are still strong enough to solve consensus, the above notion of timeliness was later refined by considering the timeliness of each pair of processes *individually*. In particular, for shared memory systems, one can define the concept of *process timeliness*, which compares the speed of a *single* process p to the speed of another process q, as follows: p *is timely with respect to* q if, for some integer i, every interval that contains i steps of q contains at least one step of p [3]. Process timeliness, however, cannot be used to study problems that are weaker than consensus such as *set agreement*: the existence of a single process p that is timely with respect to another process q is sufficient to solve consensus in read-write shared memory systems where at most one process may crash (this follows from results in [1, 3]). In fact, all the partially synchronous systems that were previously proposed for message-passing and read-write shared memory are strong enough to solve consensus (under some condition on the number of processes that may crash).

In this paper, we propose a simple generalization of process timeliness, called *set timeliness*, and show that it can be used to study and compare the partial synchrony requirements of problems that are weaker than consensus. Intuitively, this generalization is obtained by considering a set of processes P in the system as a *single entity*, i.e., as a "virtual process" p that takes a step whenever any process in P takes a step, and then use the definition of process timeliness on such virtual processes. So, a *set* of processes P is timely with respect to another *set* of processes Q if, for some integer i, every interval that contains i steps of processes in Q contains at least one step of some process in P. As we will see below, the processes in P may not be *individually* timely (i.e., the speed of each process in P may fluctuate beyond any bound), but when they are viewed as a single (cooperating) process they may be timely. So a set of processes may be able to overcome the speed fluctuations

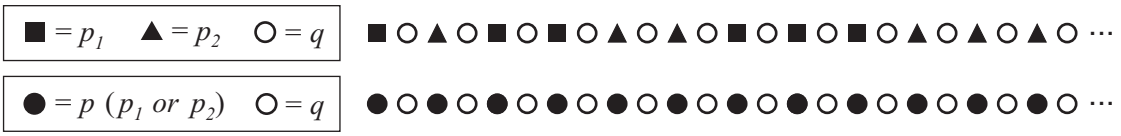

Figure 1: Example of set timeliness. Top shows a schedule with three processes, p_1, p_2, q, in which neither p_1 nor p_2 is timely with respect to q. Bottom shows the same schedule where p_1 and p_2 are considered as a single virtual process p, and p is timely with respect to q.

of individual members of the set, by working together as a timely virtual process.

A simple example, depicted in Figure 1, illustrates the definition of set timeliness. Consider the synchrony of processes p_1 and p_2 with respect to process q in schedule $S = [(p_1 \cdot q)^i \cdot (p_2 \cdot q)^i]_{i=1}^{\infty}$. Note that p_1 is *not* timely with respect to q in S, because there are longer and longer sequences of consecutive steps in S where q takes more and more steps while p_1 takes no step at all: intuitively, there are longer and longer periods where p_1 is very slow with respect to q. Similarly, p_2 is *not* timely with respect to q in S. But if we consider p_1 and p_2 as a single virtual process p, then the above schedule S now becomes $(p \cdot q)^{\infty}$, and the virtual process p is indeed timely with respect to q. In other words, if p_1 and p_2 are considered as a single entity (a set of two cooperating processes), then together they are timely with respect to q. In our model of partial synchrony, we say that the set of processes $\{p_1, p_2\}$ is timely with respect to the set $\{q\}$. Similarly, a set of processes $\{p_1, p_2\}$ is timely with respect to a set $\{q_1, q_2, q_3\}$ if, when we remove all the indices from these processes, the resulting virtual process p is timely with respect to virtual process q.

In this paper, we show that set timeliness can be used to find partially synchronous systems for sub-consensus tasks. In particular, we use it to define the first partially synchronous system for read/write shared memory that separates consensus and set agreement: we prove that this system is timely enough for solving set agreement but not enough for solving consensus. Set timeliness also allows us to define a family of partially synchronous systems of n processes, denoted $\mathcal{S}_n^k$ ($1 \leq k \leq n-1$), which closely matches the family of k-anti-Ω failure detectors that were shown to be the weakest failure detectors for the *k-set agreement* problem [4,7,12]: we prove that for $1 \leq k \leq n-1$, $\mathcal{S}_n^k$ is synchronous enough to implement k-anti-Ω but not enough to implement $(k-1)$-anti-Ω. Note that, since k-set agreement can be solved with k-anti-Ω, this implies that it can be solved in system $\mathcal{S}_n^k$.

The definition of system $\mathcal{S}_n^k$ is very simple: it is a read/write shared memory system of n processes where, in every schedule, there is at least one set of processes of size k that is timely with respect to the set of all processes in the system.

Our work is related to results in the IIS and IRIS models [11, 20, 21]. We discuss this and other related work in Section 8.

Roadmap. This paper is organized as follows. In Section 2, we define the notion of set timeliness and some partially synchronous models that are based on it. In Section 3, we describe the problems and failure detectors that we consider. In Section 4, we show that consensus cannot be solved in system $\mathcal{S}_n^{n-1}$. In Section 5, we explain why set agreement is solvable in system $\mathcal{S}_n^{n-1}$. In Section 6, we show that failure detector k-anti-Ω can be implemented in system $\mathcal{S}_n^k$. In Section 7, we show that k-anti-Ω cannot be implemented in system $\mathcal{S}_n^{k+1}$. In Section 8, we discuss related work.

2. MODEL

We consider a shared-memory system with n processes $\Pi_n = \{1, \ldots, n\}$, which can communicate with each other via some (possibly infinite) set Ξ of shared registers.

A *schedule* S (in Π_n) is a finite or infinite sequence of processes (in Π_n). A *step of a schedule* is an element of S. Given a finite schedule S and a schedule S', we denote by $S \cdot S'$ the concatenation of S and S'. Given an infinite schedule S, a process p is *correct in* S if there are infinitely many occurrences of p in S.

2.1 Set timeliness

In what follows, A and B are sets of processes in Π_n and S is a schedule in Π_n.

Definition 1 *A is timely with respect to B in S if there is an integer i such that every sequence of consecutive steps of S that contains i occurrences of processes in B contains a process in A.*

Definition 2 *A is timely in S if A is timely with respect to Π_n in S.*

From the above definitions, we have the following:

Observation 3 *A is timely in S if and only if there is an integer i such that every sequence of i consecutive steps in S contains a process in A.*

The definition of *set* timeliness given above (Definition 1) is a direct generalization of the definition of *process* timeliness given in [3]. In fact, Definition 1 can be used to define process timeliness: A process *p is timely with respect to a process q in S* if set $\{p\}$ is timely with respect to set $\{q\}$ in S.

2.2 Systems and partially synchronous systems

A system may be defined by some properties, e.g., timeliness properties, of its schedules. So we define a system $\mathcal{S}$ as a tuple $\mathcal{S} = (\Pi_n, \Xi, \mathit{Scheds})$ where *Scheds* is a set of schedules; intuitively, *Scheds* is the set of schedules that are possible in system $\mathcal{S}$.

In this paper we consider two families of partially synchronous systems: for each $1 \leq k \leq n$, $\mathcal{S}_n^k$ is the system where *at least one* set of processes of size k is timely, and $\overline{\mathcal{S}}_n^k$ is the system where *all* the sets of processes of size k are timely.[1] More precisely, for every $1 \leq k \leq n$, let Scheds_n^k be the set of schedules S in Π_n such that *at least one* set of processes of size k in Π_n is timely in S. Similarly, for every $1 \leq k \leq n$, let $\overline{\mathit{Scheds}}_n^k$ be the set of schedules S in Π_n such that *every* set of processes of size k in Π_n is timely in S. We define $\mathcal{S}_n^k = (\Pi_n, \Xi, \mathit{Scheds}_n^k)$ and $\overline{\mathcal{S}}_n^k = (\Pi_n, \Xi, \overline{\mathit{Scheds}}_n^k)$.

[1] To strengthen our results, we will use $\mathcal{S}_n^k$ when we give algorithms and $\overline{\mathcal{S}}_n^k$ when we prove impossibility results.

2.3 Algorithms and runs

An algorithm $\mathcal{A}$ in a system $\mathcal{S}$ consists of a set of n (infinite or finite) deterministic automata $\mathcal{A}_1, \ldots, \mathcal{A}_n$. By abuse of notation, we identify a process with its automaton. Each process executes by taking steps. In each step, a process p can read or write a shared register and change state (according to p's state transition function in $\mathcal{A}_p$).

Below, $\mathcal{A}$ denotes an algorithm, $\mathcal{S} = (\Pi_n, \Xi, \mathit{Scheds})$ denotes a system, and *pref*(*Scheds*) denotes the set of all finite prefixes of schedules in *Scheds*. A *configuration* of $\mathcal{A}$ in $\mathcal{S}$ indicates the state of each process and register. A *run* R of $\mathcal{A}$ in $\mathcal{S}$ is a tuple $R = (I, S, \mathcal{A})$ where I is an initial configuration of $\mathcal{A}$ in $\mathcal{S}$ and S is a schedule in *Scheds*. A *partial run* P of $\mathcal{A}$ in $\mathcal{S}$ is a tuple $P = (I, S, \mathcal{A})$ where I is an initial configuration of $\mathcal{A}$ in $\mathcal{S}$ and S is a schedule in *pref*(*Scheds*). The *configuration at the end of* P is the state of each process and register after they have taken steps from I in the order indicated by S and according to the state transitions of $\mathcal{A}$. Given a schedule S' where $S \cdot S' \in \mathit{pref}(\mathit{Scheds})$, we denote by $P \cdot S'$ the partial run $(I, S \cdot S', \mathcal{A})$ of $\mathcal{A}$ in $\mathcal{S}$. A *continuation of* P *in* $\mathcal{S}$ is a run $R = (I, S', \mathcal{A})$ of $\mathcal{A}$ in $\mathcal{S}$ where S is a prefix of S'.

3. PROBLEMS AND FAILURE DETECTORS

3.1 Consensus, set agreement, and k-set agreement

In the *k-set agreement* problem (for $1 \leq k \leq n$), each process has an initial value and must decide on a value such that

- *(Uniform k-agreement)* Processes decide on at most k distinct values;
- *(Uniform validity)* If some process decides on v then v is the initial value of some process; and
- *(Termination)* Every correct process eventually decides on some value.

The $(n-1)$-set agreement problem is also called simply the *set agreement problem*. The 1-set agreement problem is also called the *consensus problem*. The *binary consensus problem* is the consensus problem such that the initial values of processes are in $\{0, 1\}$.

3.2 Failure detectors anti-Ω and k-anti-Ω

Failure detectors [6] are basic tools of fault-tolerant distributed computing that can be used to solve fundamental problems such as consensus, set agreement, and atomic broadcast. In this paper, we consider the family of *k-anti-*Ω failure detectors [22] for $1 \leq k \leq n$. With k-anti-Ω, every process p has a local variable $\mathit{fdOutput}_p$ that holds a set of $n-k$ processes, such that the following property holds: if some process is correct then there exists a correct process c and a time after which, for every correct process p, c is not in $\mathit{fdOutput}_p$. Note that $(n-1)$-anti-Ω is also called simply *anti-*Ω, and 1-anti-Ω is equivalent to the Ω failure detector [6].

4. IMPOSSIBILITY OF CONSENSUS IN SYSTEM $\mathcal{S}_n^{n-1}$

We now show that consensus cannot be solved in $\mathcal{S}_n^{n-1}$—the partially synchronous system of n processes where at least one set of processes of size $n-1$ is timely. In fact, we prove that consensus cannot be solved even if *every* set of processes of size $n-1$ is timely, i.e., it cannot be solved even in $\overline{\mathcal{S}}_n^{n-1}$.[2]

[2]We show this impossibility result for every $n \geq 3$. For $n = 2$, consensus *is* solvable in $\mathcal{S}_2^1$ [3].

We now explain the general idea of how we prove this result. The proof is a modification of the well-known proof of impossibility of consensus in asynchronous systems [10,16]. That proof considers a purported algorithm $\mathcal{A}$ for consensus and then constructs a run R_{bad} of $\mathcal{A}$ in which processes never decide—a contradiction that shows that $\mathcal{A}$ does not exist. The construction of R_{bad} uses a bivalency argument. Roughly speaking, a partial run is *bivalent* if it has two continuations in which processes decide different values. Note that, in a bivalent partial run, no process has decided. Moreover, one can show that (a) there is some bivalent initial partial run P_0 of $\mathcal{A}$, and (b) a bivalent partial run of $\mathcal{A}$ can be extended one step at a time into increasingly larger bivalent partial runs. In the limit, we obtain an infinite run R_{bad} in which processes never decide.

The construction of [10, 16], however, does not work to show our result. This is because R_{bad} may not belong to $\overline{\mathcal{S}}_n^{n-1}$. To belong to $\overline{\mathcal{S}}_n^{n-1}$, we would like R_{bad} to have some synchrony. In particular, we must construct R_{bad} so that every set of size $n-1$ is timely. However, we cannot aim for too much synchrony, otherwise consensus becomes solvable—for instance, there are consensus algorithms in the partially synchronous model of [1]. Thus, in constructing R_{bad}, we need to find the right balance of timeliness.

We achieve this balance via the notion of a *move*. A move is a schedule of two steps by distinct processes (in later sections we extend this notion), and we consider runs that consist of finite or infinite sequences of moves such that the second process in a move is different from the first process in a subsequent move. We show that in such runs, every set of processes of size $n-1$ is timely, so these runs are in $\overline{\mathcal{S}}_n^{n-1}$. Moreover, it turns out that we can still carry out the bivalency argument using these runs alone: there exists some bivalent initial partial run and this run can be extended one move at a time, while keeping the run bivalent. This gives us the needed bad run R_{bad} in $\overline{\mathcal{S}}_n^{n-1}$.

We now state the precise result and give the detailed proof.

Theorem 4 *For every $n \geq 3$, there is no algorithm that solves binary consensus in $\overline{\mathcal{S}}_n^{n-1}$.*

We prove Theorem 4 by contradiction. Let $n \geq 3$ and suppose that there is an algorithm $\mathcal{A}$ that solves binary consensus in $\overline{\mathcal{S}}_n^{n-1}$.

We now consider the behavior of $\mathcal{A}$ in a system, denoted $\overline{\mathcal{S}}$, in which no consecutive steps are of the same process. We first show that all runs of $\mathcal{A}$ in $\overline{\mathcal{S}}$ are also in $\overline{\mathcal{S}}_n^{n-1}$. We then show that there is a run of $\mathcal{A}$ in $\overline{\mathcal{S}}$ such that processes never decide—which is a contradiction.

Definition 5

- $\overline{\mathit{Scheds}} = \{S \mid S$ *is a schedule in* Π_n *such that* S *has no two consecutive steps by the same process* $\}$.
- $\overline{\mathcal{S}}$ *is the system* $\overline{\mathcal{S}} = (\Pi_n, \Xi, \overline{\mathit{Scheds}})$.

Observation 6 *For every infinite schedule $S \in \overline{\mathit{Scheds}}$, there are at least two correct processes in S.*

Lemma 7 *For every schedule $S \in \overline{\mathit{Scheds}}$, every set of processes of size $n-1$ is timely in S.*

PROOF. Consider a schedule $S \in \overline{\mathit{Scheds}}$, and let A be a set of processes of size $n-1$ in Π_n. By the definition of $\overline{\mathit{Scheds}}$, every two consecutive steps of S contain at least one process in A. By Observation 3, A is timely in S. □

Lemma 7 immediately implies the following:

Corollary 8 $\overline{Scheds} \subseteq \overline{Scheds}_n^{n-1}$.

Corollary 9 *Every run of algorithm $\mathcal{A}$ in system $\overline{\mathcal{S}}$ is also a run of $\mathcal{A}$ in system $\overline{\mathcal{S}}_n^{n-1}$.*

We now define the standard notion of bivalent and v-valent partial runs *with respect to the runs of algorithm $\mathcal{A}$ in system $\overline{\mathcal{S}}$*.

Definition 10 *Let P be a partial run of algorithm $\mathcal{A}$ in system $\overline{\mathcal{S}}$ and $v \in \{0,1\}$. P is* v-valent *in $\overline{\mathcal{S}}$ if, in every continuation of P in $\overline{\mathcal{S}}$, no process decides a value other than v. P is* bivalent *in $\overline{\mathcal{S}}$ if it is not v-valent in $\overline{\mathcal{S}}$ for any $v \in \{0,1\}$.*

Note that if a partial run P is bivalent then there are continuations C_0 and C_1 of P such that some process decides 0 in C_0 and some process decides 1 in C_1.

Henceforth in this section, a (partial) run refers to a (partial) run of the algorithm $\mathcal{A}$ in system $\overline{\mathcal{S}}$. Moreover, when we say bivalent or v-valent, for some $v \in \{0,1\}$, we mean bivalent or v-valent in system $\overline{\mathcal{S}}$.

Definition 11 *Given two partial runs P and P', we say that P* and P' are indistinguishable by a process p *if p has the same state in the configurations at the end of P and P'. We say that P* and P' are indistinguishable by the shared registers *if all shared registers have the same state in the configurations at the end of P and P'.*

Lemma 12 *There exists a partial run P that is bivalent.*

PROOF. (Similar to a proof in [10]) Suppose by contradiction that every partial run is v-valent for some v. Consider the partial runs $P_0, P_1, \ldots, P_n$ with empty schedule such that, in P_i, the initial value of processes $0, \ldots, i$ is 0 and the initial value of processes $i+1, \ldots, n$ is 1. Then P_0 is 0-valent and P_n is 1-valent. Since for every i, P_i is either 0-valent or 1-valent, we can find some $j \leq n-1$ such that P_j is 0-valent and P_{j+1} is 1-valent. The initial configurations of P_j and P_{j+1} only differ in the state of process $j+1$. Let p and q be distinct processes different from $j+1$ (p and q exist because $n \geq 3$) and consider the continuations C_0 and C_1 of P_j and P_{j+1}, respectively, in which p and q keep taking steps in alternation (forever). Note that C_0 and C_1 are runs in $\overline{\mathcal{S}}$ because no two consecutive steps are of the same process. It is easy to see that p and q go through the same sequence of states in C_0 and C_1, and therefore they decide the same value in both continuations. However, they decide 0 in C_0 since P_j is 0-valent, and they decide 1 in C_1 since P_{j+1} is 1-valent—a contradiction. □

We now define the notion of a *move* and then use sequences of moves to obtain runs in system $\overline{\mathcal{S}} = (\Pi_n, \Xi, \overline{Scheds})$.

Definition 13 *A* move m *consists of an ordered pair of processes (p,q) with $p \neq q$.*

Observation 14 *Any finite or infinite sequence of moves $m_o m_1 \ldots$ such that the second process of each move is different from the first process of the subsequent move is a schedule in $\overline{Scheds}$.*

Definition 15 *Given a partial run P, we say that* move $m = (p,q)$ is applicable to P *if the schedule of P is empty or if the last process of P is different from p.*

By Observation 14, we have

Observation 16 *By successively applying (applicable) moves to a partial run in $\overline{\mathcal{S}}$, we obtain a run in $\overline{\mathcal{S}}$.*

Lemma 17 *If a partial run P is bivalent then there exists a move m applicable to P such that partial run $P \cdot m$ is bivalent.*

PROOF. Let P be a bivalent partial run. Suppose, by contradiction, that for every move m applicable to P, $P \cdot m$ is v-valent for some v.

Since P is bivalent, there are two moves m_0 and m_1 such that $P \cdot m_0$ is 0-valent and $P \cdot m_1$ is 1-valent.

Consider the following cases:

Case 1: $m_0 = (p,q)$ and $m_1 = (p,q')$ for some p, q, and q'. First, note that $p \neq q$, $p \neq q'$ and $q \neq q'$. Now consider the step that q and q' take when m_0 and m_1 are applied to P. There are two cases:

Case 1.1: q and q' operate on different shared registers. Let $P_0 = P \cdot (p,q) \cdot (q',p)$ and $P_1 = P \cdot (p,q') \cdot (q,p)$. Then P_0 and P_1 are indistinguishable by every process and by the shared registers, and P_0 is 0-valent while P_1 is 1-valent. Consider a continuation C_0 of P_0 in which the move (q,p) occurs repeatedly (and no other). Consider a continuation C_1 of P_1 in which the move (q,p) occurs repeatedly (and no other). In both continuations, processes q and p go through the same set of states, and so they must decide the same value in both continuations. However, since P_0 is 0-valent, in C_0 processes p and q decide 0, and since P_1 is 1-valent, in C_1 processes p and q decide 1—a contradiction.

Case 1.2: q and q' operate on the same shared register V. We consider the following sub-cases:

- If both q and q' read V then let $P_0 = P \cdot (p,q) \cdot (q',p)$ and $P_1 = P \cdot (p,q') \cdot (q,p)$. Then P_0 and P_1 are indistinguishable by every process and by the shared registers, and P_0 is 0-valent while P_1 is 1-valent. We reach a contradiction as above.
- If q reads V and q' writes V then let $P_0 = P \cdot (p,q) \cdot (q',p)$ and $P_1 = P \cdot (p,q') \cdot (q,p)$. Then P_0 and P_1 are indistinguishable by every process other than q and by the shared registers, and P_0 is 0-valent and P_1 is 1-valent. We reach a contradiction as above by considering continuations in which the move (q',p) occurs repeatedly.
- If q writes V and q' reads V, this case is analogous to the one in which q reads V and q' writes V.
- If both q and q' write to V, consider the continuation C_0 of $P \cdot (p,q)$ in which move (q',p) occurs repeatedly, and the continuation C_1 of $P \cdot (p,q')$ in which the move (p,q') occurs repeatedly. The state of q' at the end of $P \cdot (p,q)$ is the same as in the end of $P \cdot p$. Therefore, when q' takes its first step after $P \cdot (p,q)$ in C_0, q' writes to V the same value that it writes after q' takes its first step after $P \cdot p$ in C_1. Thus, it is easy to see that p and q' go through the same sequence states in C_0 and C_1, and therefore they decide the same value in both continuations. However, in C_0 they decide 0 and in C_1 they decide 1—a contradiction.

Case 2: $m_0 = (p, q)$ *and* $m_1 = (p', q')$ *for some p, q, and* q' *such that* $p \neq p'$. Note that $p \neq q$ and $p' \neq q'$.

By assumption, $P \cdot (p, p')$ is 0-valent or 1-valent. If $P \cdot (p, p')$ is 1-valent then, since $P \cdot (p, q)$ is 0-valent, we fall back to Case 1, which leads to a contradiction. So $P \cdot (p, p')$ is 0-valent.

By assumption, $P \cdot (p', p)$ is 0-valent or 1-valent. If $P \cdot (p', p)$ is 0-valent then, since $P \cdot (p', q')$ is 1-valent, we fall back to Case 1, which leads to a contradiction. So $P \cdot (p', p)$ is 1-valent.

Now consider the step that p and p' take when (p, p') and (p', p), respectively, are applied to P. There are two cases:

Case 2.1: p *and* p' *operate on different shared registers.* Let $P_0 = P \cdot (p, p')$ and $P_1 = P \cdot (p', p)$. Then P_0 and P_1 are indistinguishable by every process and by the shared registers, and P_0 is 0-valent while P_1 is 1-valent. Let r be a process different from p and p'. Consider two continuations C_0 and C_1 of P_0 and P_1, respectively, in which the move (r, p) occurs repeatedly. In both continuations, r and p go through the same sequence of states, and so they decide the same value. However, since P_0 is 0-valent, in C_0 processes r and p decide 0 and and since P_1 is 1-valent, in C_1 processes r and p decide 1—a contradiction.

Case 2.2: p *and* p' *operate on the same shared register* V. We consider the following sub-cases:

- If both p and p' read V then let $P_0 = P \cdot (p, p')$ and $P_1 = P \cdot (p', p)$. Then P_0 and P_1 are indistinguishable by every process and by the shared registers, and P_0 is 0-valent while P_1 is 1-valent. We reach a contradiction as above.
- If p reads V while p' writes V then let $P_0 = P \cdot (p, p')$ and $P_1 = P \cdot (p', p)$. Then P_0 and P_1 are indistinguishable from every process other than p and by the shared registers, and P_0 is 0-valent and P_1 is 1-valent. We reach a contradiction as above by considering continuations of P_0 and P_1 in which the move (r, p') occurs repeatedly, where r is a process other than p and p'.
- If p writes V while p' reads V, this case is analogous to the one in which p reads V and p' writes V.
- If both p and p' write V, let r be a process other than p and p'. Consider the two partial runs $P \cdot (p, p')$ and $P \cdot (p, r)$. Recall that $P \cdot (p, p')$ is 0-valent. By assumption, $P \cdot (p, r)$ is 0-valent or 1-valent. If it is 1-valent then we fall back to Case 1, which leads to a contradiction. So $P \cdot (p, r)$ is 0-valent. Let C_0 be a continuation of $P \cdot (p, r)$ in which the (p, r) occurs repeatedly. Let C_1 be a continuation of $P \cdot (p', p)$ in which the move (r, p) occurs repeatedly. The last step of p in $P \cdot (p', p)$ causes the same value to be written to V as the last step of p in $P \cdot (p, r)$. Thus, it is easy to see that p and r go through the same sequence of states in C_0 and C_1, and therefore they decide the same value in both continuations. However, $P \cdot (p, r)$ is 0-valent so p and r decide 0 in C_0, and $P \cdot (p', p)$ is 1-valent so p and r decide 1 in C_1—a contradiction.

□

The following corollary states the contradiction that concludes the proof of Theorem 4:

Corollary 18 *There is an infinite run in which processes never decide.*

PROOF. From Lemmas 12 and 17, and Observation 16. □

Since the partially synchronous system $\mathcal{S}_n^{n-1}$ is weaker than $\overline{\mathcal{S}}_n^{n-1}$ (i.e., the set of schedules of $\mathcal{S}_n^{n-1}$ is a superset of the set of schedules of $\overline{\mathcal{S}}_n^{n-1}$), Theorem 4 implies the following:

Theorem 19 *For every* $n \geq 3$, *there is no algorithm that solves binary consensus in* $\mathcal{S}_n^{n-1}$.

As we mentioned earlier, this impossibility result does not hold for the special case $n = 2$, because in this case it is possible to implement Ω [2] and with Ω one can solve consensus in the shared memory model [15]:

Observation 20 *There is an algorithm that solves consensus in* $\mathcal{S}_2^1$.

5. POSSIBILITY OF SET AGREEMENT IN SYSTEM $\mathcal{S}_n^{n-1}$

In the next section, we give an algorithm that implements $(n-1)$-anti-Ω in system $\mathcal{S}_n^{n-1}$ (see Theorem 31). Since $(n-1)$-anti-Ω can be used to solve set agreement [22],[3] the following corollary of Theorem 31 is immediate:

Corollary 21 *For every* $n \geq 2$, *set agreement can be solved in system* $\mathcal{S}_n^{n-1}$.

6. POSSIBILITY OF k-ANTI-Ω IN SYSTEM $\mathcal{S}_n^k$

We now give an algorithm for k-anti-Ω for system $\mathcal{S}_n^k$, that is, the algorithm works if every run has at least one timely set of size k. In the next section, we show that there are no algorithms for $(k-1)$-anti-Ω for system $\mathcal{S}_n^k$. Therefore, in a precise sense, we establish a tight correspondence between the set of partially synchronous systems $\{\mathcal{S}_n^1, \ldots, \mathcal{S}_n^n\}$ and the set of failure detectors $\{1\text{-anti-}\Omega, \ldots, n\text{-anti-}\Omega\}$ for k-set agreement.

In the following, $\mathbf{\Pi}_n^k$ denotes the set of all subsets of Π_n of size k. The basic idea of our algorithm is that each process has a heartbeat that it increments periodically, and each process has a timeout timer on each set A in $\mathbf{\Pi}_n^k$. A process resets the timer for A whenever it sees that the heartbeat of some process in A has increased. If the timer for A expires (the process times out on A), the process increments a shared register *Counter*$[A]$, which represents a "badness" counter for A, and it also increments the timeout that it subsequently uses for A. The process picks the set *winnerset* with smallest counter, breaking ties using some arbitrary total order on $\mathbf{\Pi}_n^k$, and then outputs the set $\Pi_n - winnerset$.

The detailed algorithm is shown in Figure 2. Each process executes an infinite loop, in which the process reads *Counter*$[A]$ for each set A in $\mathbf{\Pi}_n^k$, chooses a winner, and sets the output of k-anti-Ω accordingly. The process then increments its heartbeat, checks the heartbeats of each process q and, if the heartbeat has increased, it resets the timers of all the sets in $\mathbf{\Pi}_n^k$ containing q. Finally, the process checks if any of the timers has expired, and if so, it increments the counter of the appropriate set.

Intuitively, this algorithm works because there is at least one set of $n - k$ processes that is timely. Therefore there is at least one set such that processes stop timing out on it—and therefore its counter eventually stops changing. Among all such sets, one of them say

[3] In fact, $(n-1)$-anti-Ω, which is simply called anti-Ω, is the weakest failure detector to solve set agreement [22].

SHARED REGISTERS

$\forall p \in \Pi_n$: *Heartbeat*$[p] = 0$
$\forall A \in \mathbf{\Pi}_n^k$: *Counter*$[A] = 0$ { $\mathbf{\Pi}_n^k$ is the set of all subsets of Π_n of size k }

CODE FOR PROCESS p:

Local variables

fdOutput = any set of processes of size $n-k$
winnerset $= \emptyset$
myHb $= 0$
$\forall q \in \Pi_n$: *prevHeartbeat*$[q] = 0$
$\forall A \in \mathbf{\Pi}_n^k$: *timeout*$[A] = 1$
$\forall A \in \mathbf{\Pi}_n^k$: *timer*$[A]$ = *timeout*$[A]$
$\forall A \in \mathbf{\Pi}_n^k$: *cnt*$[A] = 0$
hbq $= 0$

Main code

```
1  repeat forever
       { choose FD output }
2      for each A ∈ Π_n^k do cnt[A] ← read(Counter[A])
3      winnerset ← argmin_{A∈Π_n^k}{(cnt[A], A)} { break ties using
                                                   a total order on Π_n^k }
4      fdOutput ← Π_n − winnerset

       { bump heartbeat }
5      myHb ← myHb + 1
6      write(Heartbeat[p], myHb)

       { check other processes' heartbeat }
7      for each q ∈ Π_n do
8          hbq ← read(Heartbeat[q])
9          if hbq > prevHeartbeat[q] then
10             for each A ∈ Π_n^k do
11                 if q ∈ A then timer[A] ← timeout[A]
12             prevHeartbeat[q] ← hbq

       { check for expiration of set timers }
13     for each A ∈ Π_n^k do
14         timer[A] ← timer[A] − 1
15         if timer[A] = 0 then
16             timeout[A] ← timeout[A] + 1
17             timer[A] ← timeout[A]
               { increment Counter[A] based on the
                   value read in line 2 }
18             write(Counter[A], cnt[A] + 1)
```

Figure 2: Implementation of k-anti-Ω in system $\mathcal{S}_n^k$ for $1 \leq k \leq n-1$.

A_0 ends up with the smallest counter, and eventually all correct processes pick this set as the winner and output $\Pi_n - A_0$. Note that A_0 must have a correct process for otherwise correct processes keep timing out on A_0 and so its counter would grow to infinity.

We now sketch a correctness proof. Henceforth, we consider an arbitrary run R of the algorithm of Figure 2, in system $\mathcal{S}_n^k$. In the proof, the local variable var of a process p is denoted by var_p. Let S be the schedule of run R. All the steps refer to steps in S, and a "correct process" refers to a correct process in S.

If there are no correct processes then all the requirements of k-anti-Ω are satisfied. Thus we assume without loss of generality that some process is correct. Since R is a run in system $\mathcal{S}_n^k$, we can define the following:

Definition 22 *Let A' be some timely set of size k.*

Lemma 23 *For all $A \in \mathbf{\Pi}_n^k$, if A is timely in S then there exists a constant c such that every c consecutive steps in S contains a step of a process in A that writes in line 6.*

PROOF SKETCH. Each loop interaction has a bounded number of steps, so the result follows from the definition of what it means for set A to be timely in S. □

Note that *Counter*$[A]$ is not monotonically increasing (it may decrease) because it is read and later written concurrently by different processes. However, we can show the following:

Lemma 24 *For all $A \in \mathbf{\Pi}_n^k$, either Counter$[A]$ stops changing or it grows without bounds.*[4]

PROOF SKETCH. For all $A \in \mathbf{\Pi}_n^k$, if *Counter*$[A]$ does not stop changing then some process writes *Counter*$[A]$ infinitely often with increasing values in line 18. Therefore it grows without bounds. □

Lemma 25 *For all $A \in \mathbf{\Pi}_n^k$, if A is timely in S then there is a time after which Counter$[A]$ stops changing.*

PROOF SKETCH. From Lemma 23, there exists a constant c such that, during the execution of any c consecutive steps, *Heartbeat*$[p]$ is incremented for some $p \in A$. Therefore, for every correct process p, there exists a constant c' such that $timer_p[A]$ is reset to $timeout_p[A]$ at least once every c' steps of p. Thus, there is a time after which p does not find that $timer_p[A] = 0$ in line 15, and so there is a time after which *Counter*$[A]$ stops changing. □

Lemma 26 *For all $A \in \mathbf{\Pi}_n^k$, if every process in A crashes then Counter$[A]$ grows without bounds.*

PROOF SKETCH. By assumption, some process p is correct. If every process in A crashes then eventually no process in A increments its entry in the *Heartbeat* vector. Thus, there is a time after which p does not set $timer_p[A]$ to $timeout_p[A]$ in line 17. Then p will write *Counter*$[A]$ infinitely often in line 18. Therefore *Counter*$[A]$ grows without bounds. □

Definition 27 *For all $A \in \mathbf{\Pi}_n^k$, let $c(A)$ be defined as follows. If Counter$[A]$ eventually stops changing then $c(A)$ is the final value of Counter$[A]$; if Counter$[A]$ grows without bounds then $c(A) = \infty$. Let $A_0 = argmin_A\{(c(A), A)\}$.*

[4] We say that x grows without bounds if for every $M \geq 0$ there is a time after which $x \geq M$.

Lemma 28 *Eventually Counter$[A_0]$ stops changing.*

PROOF. We assume that set A' is timely and so, by Lemma 25, *Counter*$[A']$ stops changing. Thus, $c(A') < \infty$. So, by the definition of A_0, $c(A_0) \leq c(A') < \infty$. Therefore, *Counter*$[A_0]$ stops changing. □

Lemma 29 *A_0 has a correct process.*

PROOF SKETCH. Immediate from Lemmas 28 and 26. □

Lemma 30 *There is a time after which all correct processes output $\Pi_n - A_0$.*

PROOF SKETCH. Let p be any correct process. It is clear that there is a time after which p can only pick A_0 in line 3, because if $A \neq A_0$ then by Lemma 24, either (a) *Counter*$[A]$ grows without bounds so that, by Lemma 28, eventually (*Counter*$[A], A) >$ (*Counter*$[A_0], A_0$) or (b) *Counter*$[A]$ stops changing so that, by definition of A_0, eventually (*Counter*$[A], A) >$ (*Counter*$[A_0], A_0$). □

Theorem 31 *For every n and k such that $1 \leq k \leq n-1$, the algorithm in Figure 2 implements k-anti-Ω in system $\mathcal{S}_n^k$.*

PROOF SKETCH. It is clear that the output at each process is a set of $n - k \geq 1$ processes. By Lemmas 29 and 30, there is a correct process c in A_0 and a time after which the set output by each correct process does not contain c. Hence all the requirements of k-anti-Ω are satisfied. □

7. IMPOSSIBILITY OF k-ANTI-Ω IN SYSTEM $\mathcal{S}_n^{k+1}$

We now show that there are no algorithms that implement k-anti-Ω in $\mathcal{S}_n^{k+1}$. In fact, we show a stronger result, namely, that there are no algorithms that implement k-anti-Ω in $\overline{\mathcal{S}}_n^{k+1}$. Our proof is based on a new and simple technique called *scheduling diagonalization*, which resembles diagonalization in Mathematics. The general idea is that we consider a candidate algorithm $\mathcal{A}$ for k-anti-Ω, and we construct a "bad" run R of $\mathcal{A}$ in $\overline{\mathcal{S}}_n^{k+1}$ where processes are scheduled according to the output of $\mathcal{A}$, such that R does not satisfy the required properties of k-anti-Ω. To ensure that R is in $\overline{\mathcal{S}}_n^{k+1}$, it is defined as an infinite sequence of moves, where each move has steps by $n-k$ processes. Therefore, a move intersects every set of processes of size $k+1$, and so every such set is timely.

To make the run R of $\mathcal{A}$ "bad", we choose the moves dynamically as we incrementally build R as a succession of moves: if p is the last process that took a step in a move of R, then the next move of R consists of the $n-k$ processes that are output by $\mathcal{A}$ at p (note that since $\mathcal{A}$ is supposed to implement k-anti-Ω , it must output a set of $n-k$ processes at each process). By this construction, the only processes that are correct in run R (i.e., take infinitely many steps in R) are those that are output by $\mathcal{A}$ at some correct process infinitely often. In other words, in run R, *every* correct process is output by $\mathcal{A}$ at some correct process infinitely often; thus $\mathcal{A}$ violates the specification of k-anti-Ω.

Theorem 32 *For every n and k such that $1 \leq k \leq n-1$, there is no algorithm that implements k-anti-Ω in system $\overline{\mathcal{S}}_n^{k+1}$.*

---

1 $I \leftarrow$ initial configuration of $\mathcal{A}$
2 $S \leftarrow$ empty schedule
3 $B \leftarrow \{1, \ldots, n-k\}$
4 **repeat forever**
5 $\quad S \leftarrow S \cdot$ *move*(B) { S is a finite schedule}
6 $\quad P \leftarrow (I, S, \mathcal{A})$ { P is a partial run of algorithm $\mathcal{A}$}
7 $\quad p \leftarrow$ last process in the schedule S
8 $\quad C \leftarrow$ configuration of $\mathcal{A}$ at the end of partial run P
9 $\quad B \leftarrow$ *fdOutput*$_p$ of p in configuration C
{ B is a set of $n-k$ processes }

Figure 3: Algorithm for generating a bad run of a purported k-anti-Ω algorithm $\mathcal{A}$ in system $\overline{\mathcal{S}}_n^{k+1}$.

PROOF SKETCH. Let n and k be such that $1 \leq k \leq n-1$. Let $\mathcal{A}$ be a arbitrary algorithm where each process p has an *fdOutput*$_p$ variable. We will show that $\mathcal{A}$ has a run R in system $\overline{\mathcal{S}}_n^{k+1}$ such that R does not satisfy the properties of k-anti-Ω. Since $\mathcal{A}$ is chosen arbitrarily, this will prove Theorem 32.

If $\mathcal{A}$ can set *fdOutput*$_p$ to a value not in $\mathbf{\Pi}_n^{n-k}$ then it is easy to construct a run R of $\mathcal{A}$ in $\overline{\mathcal{S}}_n^{k+1}$ that violates the properties of k-anti-Ω. So, henceforth assume that *fdOutput*$_p \in \mathbf{\Pi}_n^{n-k}$ for all processes p. Run R is constructed by the algorithm in Figure 3. This algorithm uses the notation "*move*(B)" defined below:

Definition 33 *For every $B \in \mathbf{\Pi}_n^{n-k}$, move$(B)$ is the finite schedule consisting of the $n-k$ processes of B in increasing order of process id.*

Each iteration of the loop of the algorithm of Figure 3 produces increasingly larger run prefixes P. In the limit, after infinitely many loop iterations, we obtain a run R:

Definition 34 *Let $R = (I, S, \mathcal{A})$ be the run of $\mathcal{A}$ generated by the algorithm of Figure 3.*

Since each move in R has $n-k \geq 1$ processes and R has infinitely many moves, we have the following:

Observation 35 *There is a process that is correct in the schedule S of R.*

Lemma 36 *For all $A \in \mathbf{\Pi}_n^{k+1}$, A is timely in S.*

PROOF SKETCH. Let $A \in \mathbf{\Pi}_n^{k+1}$. Any $2(n-k)$ consecutive steps of S contains *move*(B) for some set B of $n-k$ processes. Since A has $k+1$ processes and B has $n-k$ processes, $A \cap B \neq \emptyset$. Thus, any $2(n-k)$ consecutive steps of S contains some process in A. By Observation 3, A is timely in S. □

By Lemma 36, the schedule S of run R is in $\overline{\mathit{Scheds}}_n^{k+1}$. Since $\overline{\mathcal{S}}_n^{k+1} = (\Pi_n, \Xi, \overline{\mathit{Scheds}}_n^{k+1})$, we have:

Corollary 37 *R is a run of $\mathcal{A}$ in system $\overline{\mathcal{S}}_n^{k+1}$.*

Lemma 38 *In run R, for every process q that is correct in S, there exists a process p that is correct in S such that $q \in$ fdOutput$_p$ infinitely often.*

PROOF SKETCH. Let q be any process that is correct in S. Since q is correct in S, q appears in the schedule S infinitely often. Thus, in the execution of the algorithm in Figure 3 that constructs run R, process q is included in the set B in line 9 infinitely often. So, since there are finitely many processes in Π_n, there must be some process p such that the following occurs infinitely often in this execution: (1) p is selected in line 7, and (2) $q \in \mathit{fdOutput}_p$ in line 9. Since p is selected infinitely often in line 7, p appears infinitely often in S, i.e., p is correct in S. □

Lemma 38 implies that R does not satisfy the required properties of k-anti-Ω, concluding the proof of Theorem 32.

It is worth noting that the construction of the "bad" run R in the above proof is not done formally in the failure detector model with failure patterns, failure detector histories, etc. However, it is easy to map our construction into a more formal one (e.g., by defining a failure pattern for the run that we constructed). □

8. RELATED WORK

Dwork, Lynch, and Stockmeyer [8] introduce the concept of partial synchrony. They propose message-passing models in which there are eventual or unknown bounds on message transmission times and on relative process speeds. These bounds must hold between every pair of processes. It is shown that consensus can be solved in these models. Subsequent work [1, 2, 9, 13, 14, 17, 19] proposed weaker types of partial synchrony (for message-passing systems) with which consensus can still be solved or Ω can be implemented (Ω is the weakest failure detector for consensus [5]). None of these works have considered models in which set agreement can be solved, but consensus cannot.

[3] defines what it means for a single process p to be timely with respect to another process q and considers a shared-memory model where each process may or may not be timely (with respect to every other process in the system). In this model, the paper shows that every shared object (e.g., consensus) can be implemented such that all the timely processes access the object in a wait-free manner — a progress property called *timeliness-based wait-freedom*.

The IIS model [11] is a round-based model in which, in each round, a process atomically writes a value and obtains a snapshot of the values written by other processes in the round. In this model, set agreement and consensus are impossible. Rajsbaum et al. [20, 21] propose a family of models called IRIS that are weaker than the IIS model. This family is parameterized by a property PR_C on the snapshot values that a process can obtain in a round. This property "restricts the asynchrony" of the system, because the fact that a snapshot cannot return certain values means that the execution cannot proceed in certain ways. Specific IRIS models are given in which k-set agreement is solvable but $(k-1)$-set agreement is not, thus providing a separation between these problems. Our model of partial synchrony differs from the IRIS models in two ways. First, we express synchrony behavior directly via timeliness properties of processes, whereas the IRIS models restrict the allowable executions via properties that snapshots must satisfy. Second, our model is based on read-write shared memory, whereas the IRIS model is based on rounds with immediate snapshots. It is possible to implement these rounds in the read-write shared memory model, but it is unclear how the restricted runs of IRIS map to the timeliness properties of the shared memory model. For instance, a process that never appears in the snapshot of other processes may be a process that is actually timely in the shared memory model that implements IRIS: this process may execute at the same speed as other processes but always start a round a few steps later.

The use of moves in our bivalency proof in Section 4 can be seen as a special type of layering as defined by Moses and Rajsbaum [18]. Roughly speaking, a layering is a function that maps a state into a set of possible subsequent states that could occur in a run. It provides a framework to carry out the bivalency technique in an abstract and general way. However, to use this framework one must still find an adequate layering function, such that it generates fair runs in the model, and it satisfies some connectedness properties. Finding such a layering, and proving it is adequate, can be non-trivial. It is possible that our proof can be carried out in that framework, but we opted for a more self-contained approach.

Acknowledgements. The authors are grateful to the anonymous referees for their many helpful comments.

9. REFERENCES

[1] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Communication-efficient leader election and consensus with limited link synchrony. In *ACM Symposium on Principles of Distributed Computing*, pages 328–337, July 2004.

[2] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. On implementing Omega in systems with weak reliability and synchrony assumptions. *Distributed Computing*, 21(4):285–314, Oct. 2008.

[3] M. K. Aguilera and S. Toueg. Timeliness-based wait-freedom: a gracefully degrading progress condition. In *ACM Symposium on Principles of Distributed Computing*, pages 305–314, Aug. 2008.

[4] A. F. Anta, S. Rajsbaum, and C. Travers. Weakest failure detectors via an egg-laying simulation (preliminary version). Tech Report RoSaC-2009-2, Grupo de Sistemas y Comunicaciones, Universidad Rey Juan Carlos, Jan. 2009. Also appears as a brief announcement in PODC 2009.

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

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

[7] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, and A. Tielmann. The disagreement power of an adversary. Article id hal-00376981, Hyper Article en Ligne, Apr. 2009. Available at http://hal.archives-ouvertes.fr/hal-00376981.

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

[9] A. Fernández and M. Raynal. From an intermittent rotating star to a leader. Technical Report 1810, IRISA, Université de Rennes, France, Aug. 2006.

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

[11] E. Gafni. Round-by-round fault detectors: Unifying synchrony and asynchrony (extended abstract). In *ACM Symposium on Principles of Distributed Computing*, pages 143–152, June 1998.

[12] E. Gafni and P. Kuznetsov. The weakest failure detector for solving k-set agreement. In *ACM Symposium on Principles of Distributed Computing*, Aug. 2009.

[13] M. Hutle, D. Malkhi, U. Schmid, and L. Zhou. Chasing the weakest system model for implementing Ω and Consensus.

IEEE Transactions on Dependable and Secure Computing. To appear.

[14] E. Jiménez, S. Arévalo, and A. Fernández. Implementing unreliable failure detectors with unknown membership. *Information Processing Letters*, 100(2):60–63, Oct. 2006.

[15] W. K. Lo and V. Hadzilacos. Using failure detectors to solve consensus in asynchronous shared-memory systems. In *International Workshop on Distributed Algorithms*, pages 280–295, Sept. 1994.

[16] M. C. Loui and H. H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163–183, 1987.

[17] D. Malkhi, F. Oprea, and L. Zhou. Omega meets Paxos: leader election and stability without eventual timely links. In *International Conference on Distributed Computing*, volume 3724 of *LNCS*, pages 199–213. Springer Verlag, Sept. 2005.

[18] Y. Moses and S. Rajsbaum. A layered analysis of consensus. *SIAM Journal on Computing*, 31(4):989–1021, 2002.

[19] A. Mostefaoui, M. Raynal, and C. Travers. Time-free and timer-based assumptions can be combined to obtain eventual leadership. *IEEE Transactions on Parallel and Distributed Systems*, 17(7):656–666, July 2006.

[20] S. Rajsbaum, M. Raynal, and C. Travers. Failure detectors as schedulers (an algorithmically-reasoned characterization). Technical Report 1838, IRISA, Université de Rennes, France, Mar. 2007.

[21] S. Rajsbaum, M. Raynal, and C. Travers. The iterated restricted immediate snapshot model. In *International Computing and Combinatorics Conference*, volume 5092 of *LNCS*, pages 487–497. Springer, June 2008.

[22] P. Zielinski. Anti-Omega: the weakest failure detector for set agreement. In *ACM Symposium on Principles of Distributed Computing*, pages 55–64, Aug. 2008.