The Gap in Circumventing the Impossibility of Consensus

Rachid Guerraoui

Distributed Programming Laboratory, EPFL,
CH-1015, Lausanne, Switzerland
tel: +41 21 693 5272, fax: +41 21 693 7570

Petr Kuznetsov *

Max Planck Institute for Software Systems,
Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany
tel.: +49 681 9325-697, fax: +49 681 9325-299

Abstract

The impossibility of reaching deterministic consensus in an asynchronous and crash prone system was established for a weak variant of the problem, usually called weak consensus, where a set of processes need to decide on a common value in $\{0, 1\}$, so that both 0 and 1 are possible decision values. On the other hand, approaches to circumventing the impossibility focused on a stronger variant of the problem, called consensus, where the processes need to decide on one of the values they initially propose (0 or 1). This paper studies the computational gap between the two problems. We show that any set of deterministic object types that, combined with registers, implements weak consensus, also implements consensus. Then we exhibit a non-deterministic type that implements weak consensus, among any number of processes, but, combined with registers, cannot implement consensus even among two processes. In modern terminology, this type has consensus power 1 and weak consensus power ∞.

Key words: Asynchronous distributed system, consensus, weak consensus, FLP impossibility, atomic objects, determinism

* Corresponding author.
Email addresses: rachid.guerraoui@epfl.ch (Rachid Guerraoui), pkouznet@mpi-sws.mpg.de (Petr Kuznetsov).

Preprint submitted to Elsevier Science *6 October 2007*

1 Introduction

Background

A consensus protocol is a distributed algorithm that makes a set of processes decide on a common value out of two possible values: 0 or 1, where both 0 and 1 are possible decision values. In 1983, it was shown that no deterministic protocol can solve consensus in a basic distributed system model where no synchrony assumption is made (i.e., in an asynchronous system), processes can only communicate by exchanging messages, and at least one process can fail by crashing [5]. The impossibility was extended later to the shared memory model where processes could communicate through atomic objects, i.e., registers [6, 11].

Given the importance of consensus in reliable distributed computing, a lot of work has been devoted to studying abstractions that, when added to the basic distributed model, circumvent the impossibility. In particular, it was suggested to augment the system model with more sophisticated synchronization abstractions than message passing channels or registers. More precisely, the idea was to study object types that should be used, besides registers, to solve consensus among two or more processes in an asynchronous system assuming an arbitrary number of possible crashes [6]. Types like queue, test-and-set or compare-and-swap can indeed be used to do so and they are said to *implement* consensus (among a specific number of processes). It was observed that certain types could implement consensus among k processes but not among $k+1$ processes. For example, instances of type queue and registers make it possible to solve consensus among 2 processes but not among 3 processes [6]. In a sense, queue is a minimal type to implement consensus among 2 processes: 2 is also said to be the consensus power of queue. In comparison, the consensus power of type register is 1: with registers only, consensus cannot be solved among 2 processes [4, 6, 11]. At the other extreme, the consensus power of type compare-and-swap is ∞ [6]: instances of this type and registers make it possible to solve consensus among any number of processes. The notion of consensus power gives rise to a hierarchy, called the *consensus hierarchy*, with types that have low consensus power at the bottom and those that have high

consensus power at the top.

Motivation

The motivation of this work is the simple observation that the original impossibility of consensus [5,11] was stated for a *weak* variant of consensus, whereas abstractions to circumvent the impossibility have been studied with a stronger consensus variant in mind.

In a weak consensus protocol, the processes can decide any value (0 or 1), provided that there is an execution of the protocol where 0 is decided and one where 1 is decided. In the stronger variant of consensus, which is simply called consensus in the literature, the value decided must be one of the values proposed. In particular, if all processes initially propose 0 (resp. 1), the decision value must be 0 (resp. 1).

It is indeed natural to state an impossibility result on the weak variant of consensus and, when seeking abstractions that circumvent the impossibility, to consider abstractions that also solve a stronger variant of consensus. However, determining that some abstraction is, in some sense, minimal to implement (the strong variant of) consensus does not mean that the abstraction is indeed minimal to circumvent the impossibility (of weak consensus).

The motivation of this work was to determine whether the gap also exists from the object type perspective. More precisely, we address the following question: If a type implements weak consensus, does it also implement consensus? In particular, is the consensus power of a type the same as its weak consensus power?

Contributions

To address the first question, we distinguish between deterministic types and non-deterministic ones. In short, a deterministic type is one such that the output and the state that result from invoking any operation on an object of that type, performed in the absence of concurrency and failures, is uniquely determined by (a) the state of the object just before invoking the operation and (b) the operation itself.

(1) We show that, for any number of processes, any set of deterministic types that includes register and implements weak consensus, also implements consensus. In a sense, the consensus and weak consensus powers of a deterministic type are the same. Said differently, the consensus and weak consensus hierarchies, when restricted to deterministic types, are the same. To prove this result, we exploit the inherent computation power of deterministic types. In short, we observe that any protocol that solves weak consensus using objects of a deterministic type boils down to reaching a *critical state* s of some object X, such that, applying different operations to s leads to distinguishable states of X. Since X is deterministic, there is a protocol that brings X to state s. We use this observation to derive a protocol that solves another variant of consensus, named *team consensus* [14,15], which implies that the protocol also solves consensus [3, 15].

(2) We show that this is not the case with non-deterministic types. Basically, we exhibit a new non-deterministic type, which we call rambler, that implements weak consensus for an arbitrary number of processes, but cannot implement consensus even among two processes. In other words, we exhibit a non-deterministic type which has weak consensus power ∞ and consensus power 1. Type rambler is constructed in such a way that, using any number of its instances, no process can obtain any meaningful information about other processes: the instances might exhibit the very same behavior for an arbitrary sequence of invocations. Intuitively, this means that type rambler cannot implement consensus even among two processes. On the other hand, the type has some non-trivial agreement properties, and these make it possible to solve weak consensus among any number of processes using just one instance of type rambler.

Our results imply that, unlike consensus, the weak consensus abstraction is not *universal* [6]: this follows from the existence of a non-deterministic type that implements weak consensus but not consensus.

Roadmap

In Section 2, we present the system model. In Section 3, we define the consen-

sus and the weak consensus problems, as well as another variant of consensus, *team consensus* [14, 15], which is a key element of one of our proofs. In Section 4, we show that any deterministic type that implements weak consensus also implements consensus. In Section 5, we show that this is not the case with non-deterministic types. In Section 6, we conclude the paper with some general observations about the questions raised in this paper.

2 Model

The model we consider in this paper is the one of [9,10]: a set of asynchronous processes communicating through atomic shared objects. We recall below the details of the model which are relevant for our results.

Processes

We consider a set Π of $n+1$ *processes* $p_0, \ldots, p_n$ $(n \geq 1)$ that communicate using shared objects. The processes are asynchronous in the sense that we do not make any assumption about their relative speeds. The processes might fail by crashing, i.e., stop executing their steps. A process that never crashes is said to be *correct*. A process that is not correct is said to be *faulty*. We do not make any assumption about the resilience of the system, i.e. on the number of processes that can fail during the computation.

Objects and types

We assume that processes communicate via applying operations on shared objects. Every object is an instance of a *type* which is defined by a tuple (Q, Q_0, O, R, δ). Here Q is a set of *states*, $Q_0 \subseteq Q$ is a set of *initial states*, O is a set of *operations*, R is a set of *responses*, and $\delta \subseteq Q \times O \times Q \times R$ is a relation, known as the *sequential specification* of the type. We assume that every sequential specification δ is *total*: for each pair $(q, o) \in Q \times O$, there exists a pair $(q', r) \in Q \times R$ such that $(q, o, q', r) \in \delta$.

For *deterministic* types, the set of initial states is a singleton ($Q_0 = \{q_0\}$) and the sequential specification can be seen a function $\delta : Q \times O \rightarrow Q \times R$.

The sequential specification of a *non-deterministic* type carries each state and operation to a non-empty set of response and state pairs.

The deterministic register type is defined as a tuple $(Q, \{\perp\}, O, R, \delta)$ where Q is the countable set of *values* that can be stored in a register ($\perp \in Q$), $O = \{read(), write(v) \mid v \in Q\}$, $R = Q \cup \{ok\}$ and $\forall v, v' \in Q$, $\delta(v, write(v')) = (v', ok)$ and $\delta(v, read()) = (v, v)$.

We assume that shared objects are *atomic*: operations applied on an atomic object can be seen as taking place instantaneously. A *wait-free linearizable* implementation of an object type [1, 6, 8] is one example of an atomic object.

Protocols

A *protocol* is a distributed deterministic automaton that identifies the sequences of events for processes $p_0, \ldots, p_n$ and shared objects. We use a simplified form of the I/O automaton formalism [12], At any point in a protocol's execution, the state of each process is called its *local state*. The set of local states together with the states of all shared objects is called the protocol's *global state*. A computation *step* of a process is defined by the process identifier, an operation on a shared object, and the corresponding response. In an *initial state* of a protocol, every object is in an initial state specified by its type. An *execution* of a protocol is a sequence of alternating global states and steps of the processes that begins with an initial state of the protocol and respects the sequential specifications of the object types. For every local state of each process, the protocol deterministically identifies the next operation the process is going to execute. We say that an execution e of P is a *p_i-solo execution* if p_i is the only process that takes steps in e.

Schedules

A *schedule* is a (finite or infinite) sequence of identifiers of processes in Π. For a given protocol P and an initial state s, we say that a schedule σ *triggers an execution* e of P, if e begins with s and processes appear in e in the order defined by σ. Clearly, if processes access only deterministic objects, a schedule and an initial state trigger exactly one execution. On the other hand, if non-deterministic objects can be accessed, a schedule and an initial state might

trigger a number of executions.

3 Variants of consensus

Weak consensus

In a *consensus* protocol, every process initially has a *proposed* value in $\{0, 1\}$. The protocol ensures that the processes reach a common decision based on their initial states [5]. Formally, a consensus protocol ensures:

- Termination: every process that takes an infinite number of computation steps eventually decides on a value in $\{0, 1\}$;
- Agreement: no two processes decide on different values.

Clearly, there is a trivial protocol that satisfies only these two properties: every process always decides 0. To filter out such protocols, the following *non-triviality* property was defined [5]:

- Weak validity: there is an execution of the protocol in which 0 is decided and an execution in which 1 is decided.

A protocol that guarantee termination, agreement and weak validity, is said to solve *weak consensus*. (Sometimes, the problem is also called *non-trivial agreement*.) It is known that there does not exists a weak consensus protocol in an asynchronous system in the presence of at least one faulty process [5,11].

Consensus

This impossibility result of [5,11] also holds for a *consensus* protocol in which, instead of weak validity, the following property is ensured:

- Validity: any decided value is the initial value of some process.

Weak consensus is trivially reduced to consensus: any solution of consensus has an execution in which 0 is decided (e.g., when all processes propose 0) and an execution in which 1 is decided (e.g., when all processes propose 1).

Consensus solvability and initial states

A set $\mathcal{S}$ of types is said to implement (weak) consensus if there exists a (weak) consensus protocol P such that in every execution of P, processes access only objects of types in $\mathcal{S}$.

A state s of a type T is called *reachable* if, for each initial state x of T, there is a sequence of operations of T that brings x to s. We use the following observation about deterministic objects: allowing the protocol designer to initialize shared objects to any reachable states does not affect the ability of deterministic objects to solve consensus. Formally, let $\mathcal{S}$ be any set of deterministic types. We denote by $\bar{\mathcal{S}}$ the *initial-state closure* of $\mathcal{S}$, i.e., the set of all deterministic types $T' = (Q, \{q'\}, O, R, \delta)$ where q' is a reachable state of some type $T = (Q, \{q\}, O, R, \delta)$ in $\mathcal{S}$.

Lemma 1 *[3] Let $\mathcal{S}$ be any set of deterministic types. If $\bar{\mathcal{S}}$ implements consensus, then $\mathcal{S}$ implements consensus.*

Team consensus

To prove our first result (in Section 4), we use a form of consensus, *team-restricted consensus* (or simply *team consensus*) [14, 15]. Formally, a protocol P solves team consensus if there is a (known a priori) partition of Π into two non-empty teams A and B such that P solves consensus for processes in Π provided all processes on the same team propose the same value. Obviously, team consensus can be solved whenever consensus can be solved. Surprisingly, the converse is also true [14, 15]:

Lemma 2 *Let $\mathcal{S}$ be any set of types. If $\mathcal{S}$ implements team consensus, then $\mathcal{S}$ also implements consensus.*

4 Deterministic types

In this section, we show that, with respect to deterministic types, weak consensus is equivalent to consensus: any set of types that implements weak consensus

also implements consensus.

Theorem 3 *Let $\mathcal{S}$ be any set of deterministic types that includes* register*. If $\mathcal{S}$ implements weak consensus, then $\mathcal{S}$ also implements consensus.*

Proof: Let P be any protocol that solves weak consensus using objects of types in $\mathcal{S}$.

Following [5], we use a bivalency argument. A global state that is reachable by an execution of P (from now on simply a state of P) is assigned a tag $v \in \{0, 1\}$ if there is an execution of P passing through that state in which some process decides v. If a state is assigned both tags 0 and 1, it is called *bivalent.* If a state is assigned only one tag v, it is called *v-valent.* A state is univalent if it is 0-valent or 1-valent. Termination of weak consensus ensures that every state of P is either bivalent or v-valent for some $v \in \{0, 1\}$.

We proceed through the following arguments:

(1) P has a bivalent initial state [5].

(2) There exists a *critical* state of P, i.e., a bivalent state s such that every step of P applied to s results in a univalent state [5]. Suppose, to obtain a contradiction, that P has no critical state. Thus, starting from the initial bivalent state and inductively proceeding to a bivalent state reachable from it, we establish an infinite execution e of P that passes through bivalent states only. By the Agreement property of weak consensus, no process can decide in a bivalent state. Hence, no process ever decides in e — contradicting the Termination property. Thus, a critical state of P exists.

(3) Let s be any critical state of P. Consider any step of P applied to s. Since protocol P and all objects that we use are deterministic, the resulting state of P is determined by the identity of the process that takes the step.

We partition the system into two teams Π_0 and Π_1: Π_j ($j \in \{0, 1\}$) consists of the processes whose steps applied to s result in a j-valent state. Since s is bivalent, the two teams are non-empty.

Let V be the set of objects used by P. We present a protocol P' that solves *team consensus* for teams Π_0 and Π_1 using objects in V initialized to their states in s and, additionally, two multiple-writer multiple-reader registers, denoted r_0 and r_1, initialized to $\perp$. For each $j \in \{0, 1\}$, let team Π_j be associated with register r_j.

In P', every process p_i first writes its input value into its team's register and then takes *its own* steps of P starting from its state in s until p_i reaches a local state of P in which a value $j \in \{0, 1\}$ is decided. (We say that j is the value p_i obtains from P.) At this point, the process reads r_j and returns the value read.

Since processes emulate an execution of P (passing through s), the Termination property of weak consensus implies that every process that takes sufficiently many steps of P obtains a value $j \in \{0, 1\}$ from P. By the definition of s, Π_0, and Π_1, value j can only be obtained if the first step of P applied to s was taken by some process $p_k \in \Pi_j$. By protocol P', prior to taking this step, p_k has written its input value in r_j: the Validity property follows.

Suppose that all members of the same team propose the same value. Thus, no two different values can be written in the team's register, and the Agreement property of weak consensus implies that no two processes return different values. Hence, P' solves team consensus using registers and objects in V initialized to their states in s. Note that all these initial states are reachable. Thus P' solves consensus using objects in $\bar{\mathcal{S}}$, the initial-state closure of $\mathcal{S}$, i.e., assuming that all objects in $\mathcal{S}$ are initialized to certain reachable states.

Since team consensus is equivalent to consensus (Lemma 2), P' can be transformed into a solution to consensus using objects in $\bar{\mathcal{S}}$. By Lemma 1, $\mathcal{S}$ implements consensus. $\square$

5 Non-deterministic types

It turns out that some *non-deterministic* atomic objects capable of implementing weak consensus are too weak to implement consensus. To illustrate this, we introduce a new non-deterministic type which we call rambler. Through accessing objects of type rambler, no process can obtain any meaningful information about other processes: the objects might exhibit the very same behavior for an arbitrary sequence of accesses. Intuitively, this means that, combined with registers, the objects of type rambler cannot solve consensus even among two processes. On the other hand, the type is strong enough to solve weak consensus.

More precisely, type rambler is defined by the tuple $(Q, \{\perp\}, O, R, \delta)$, where:

- $Q = \{\perp, t_0, t_1, 0, 1\}$ is the set of its states;
- $O = \{o_0, o_1\}$ is the set of its operations;
- $R = \{0, 1\}$ is the set of its responses;
- and δ, its sequential specification, is

$$\delta = \{(\perp, o_i, t_j, j), (\perp, o_i, t_j, 1-j), (t_0, o_i, i, i), \\ (t_1, o_i, 1-i, 1-i), (j, o_i, j, j) \mid i, j \in \{0, 1\}\}$$

The state transition graph of a rambler object is depicted in Figure 1. The nodes of the graph define the states of the object and the edges define operations applied in the states and the corresponding responses.

Note that type rambler is built in such a way that, by accessing only objects of this type, there is no way for a process to learn anything about steps of other processes. More precisely, objects of type rambler satisfy the following property:

Lemma 4 *Let P be any protocol that uses atomic objects of types in $\{$rambler, register$\}$, s_0 be any initial state of P, and σ be any schedule. Then there is an execution e of P triggered by σ and s_0 in which every operation (if any) on an object of type rambler returns 0.*

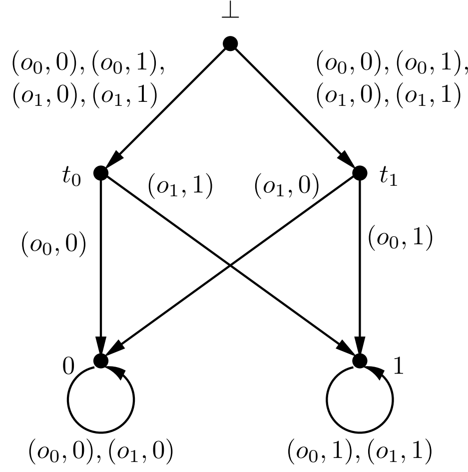

Fig. 1. State transition graph of rambler

Proof: An adversary constructs an execution triggered by σ as follows. Processes take steps according to σ until an object of type rambler is accessed for the first time. We assume that the object returns 0 (this is possible for any invocation) and the adversary does not specify its state until the object is accessed for the second time. (The objects can be in any state in $\{t_0, t_1\}$.) Assume that the operation with which the object is accessed for the second time is o_i ($i \in \{0,1\}$). Then, after the first invocation, the object has state t_i. By the specification of type rambler, the object returns 0 on the second and all subsequent invocations.

By repeating the argument for every object of type rambler, the adversary constructs an execution in which all operations on objects of type rambler return nothing but 0. □

Weak consensus with rambler

Despite the weak "synchronization power" of objects of type rambler expressed by Lemma 4, a single object of the type can implement weak consensus: p_i just invokes $o_{i \ mod \ 2}$ twice on the object and decides on the last value returned. After the first operation, the object is brought to a state in $\{t_0, t_1\}$ and becomes deterministic. Assume that the object is in state t_0. Now if p_0 is the first to access it with operation o_0, then the decision value is 0. If p_1 is the first to access it, then the decision value is 1. The case when the state of the object is t_1 is symmetric. Thus, there exist a 0-valent and a 1-valent execution, so the

Weak Validity property is ensured. The protocol returns at most one value in $\{0,1\}$ in any execution, so the Agreement is also ensured.

Impossibility of consensus with rambler

Theorem 5 *No protocol can solve consensus among 2 processes with objects of types in* $\{$rambler, register$\}$*.*

Proof: We proceed by contradiction. Let P be a protocol that solves consensus among 2 processes, p_0 and p_1, using atomic objects of types in $\{$rambler, register$\}$. Consider an execution of P in which a process p_i decides. We call the local state of p_i just after the decision the *final* state of p_i. Similar to [2,3,7], we define the *decision graph* of P [13], denoted by $\mathcal{C}(P)$, as follows. Vertices of $\mathcal{C}(P)$ represent the final states of processes p_0 and p_1 resulting from all possible executions of P (for all possible initial states). Two vertices are connected by an edge if the corresponding final states can appear in the same state of P.

We establish a contradiction through the following steps.

(1) By Lemma 4, for each schedule σ and each initial state s of P, there exists an execution $e_{\sigma,s}$ of P triggered by σ and s in which every operation on an object of type rambler returns 0. Since the protocol is deterministic, for each σ and s, there is exactly one such execution $e_{\sigma,s}$. Let $\mathcal{C}'$ be the subgraph of $\mathcal{C}(P)$ corresponding to all executions $e_{\sigma,s}$ (for all possible σ and s).

(2) For all $i \in \{0,1\}$, let v_i be the vertex of $\mathcal{C}'$ corresponding to a p_i-solo execution in which p_i proposes i and, by the Validity property of consensus, decides i. (There is a unique such vertex, since p_i is the only process to decide.) We show in the following that v_0 and v_1 belong to the same connected component of $\mathcal{C}'$.

    Assume, to obtain a contradiction, that v_0 and v_1 belongs to distinct components of $\mathcal{C}'$, $\mathcal{C}_0$ and $\mathcal{C}_1$, respectively. Then we convert P into a protocol P' that solves consensus among two processes using only registers. Every process p_i ($i \in \{0,1\}$) writes its proposed value in register r_i and then takes steps of protocol P (proposing i), except that each time p_i is about to invoke an operation on an object of type rambler, p_i updates its

state according to P *as if* the operation returned 0. By Lemma 4, for p_i, every execution of P' is indistinguishable from an execution of P in which every operation on an object of type rambler returns 0. Thus, finally, p_i ends up with a state in $\mathcal{C}'$. If the state of p_i belongs to $\mathcal{C}_0$, p_i reads r_0 and decides on the value read. Otherwise, p_i reads r_1 and decides on the value read.

The Termination property of P' follows immediately from the Termination property of P. Final states that correspond to the same execution of P belong to the same component of $\mathcal{C}'$, so the Agreement property of P' is also ensured. Note that, by the assumption, for all $i \in \{0, 1\}$, $v_{1-i} \notin C_i$. Thus, in any execution of P', no process can reach a state in $\mathcal{C}_i$ unless p_i has taken at least one step of P. By the protocol, before taking a step of P, p_i writes its input value in r_i. Thus, no process can decide on a value unless the value was proposed by some process — the Validity property is ensured.

So two processes solve consensus using only registers, contradicting [5, 11]. Therefore, v_0 and v_1 belong to the same connected component of $\mathcal{C}'$.

(3) By the Validity property, p_0 decides on 0 in v_0 and p_1 decides on 1 in v_1. Consider the path connecting v_0 and v_1 in C'. Recall that every vertex of C' denotes a final local state of some process. Any two neighbors on the path correspond to the same execution of P, and, by the Agreement property, must have the same decision value. But v_0 and v_1 have different decision values — a contradiction.

Thus, we conclude that no protocol can solve consensus among two processes using objects of types in {rambler, register}. □

6 Concluding notes

The motivation of this work was the observation that the impossibility of consensus was established for a weak variant of the problem, namely weak consensus, whereas research on circumventing the impossibility has been performed on the stronger consensus variant.

This paper exhibits the computational gap between the two problems. We show that any set of deterministic object types that, combined with registers, implements weak consensus, also implements consensus. Then we exhibit a non-deterministic type that implements weak consensus, among any number of processes, but, combined with registers, cannot implement consensus even among two processes. In modern terminology, this type has consensus power 1 and weak consensus power ∞. On the other hand, consensus is *universal* [6] and has consensus number ∞. Using consensus and registers, any type can be implemented.

Acknowledgments

We are grateful to Partha Dutta for an interesting discussion on the subject, to Eli Gafni for the insightful observation that nondeterminism in the system model may be the only reason for solving weak consensus but not consensus, and to Faith Ellen and anonymous reviewers for their helpful comments.

References

[1] H. Attiya, J. L. Welch, Distributed Computing: Fundamentals, Simulations and Advanced Topics (2nd edition), Wiley, 2004.

[2] O. Biran, S. Moran, S. Zaks, A combinatorial characterization of the distributed tasks which are solvable in the presence of one faulty processor, in: Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing (PODC), 1988.

[3] E. Borowsky, E. Gafni, Y. Afek, Consensus power makes (some) sense!, in: Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing (PODC), 1994.

[4] D. Dolev, C. Dwork, L. J. Stockmeyer, On the minimal synchronism needed for distributed consensus, Journal of the ACM 34 (1) (1987) 77–97.

[5] M. J. Fischer, N. A. Lynch, M. S. Paterson, Impossibility of distributed consensus with one faulty process, Journal of the ACM 32(3) (1985) 374–382.

[6] M. Herlihy, Wait-free synchronization, ACM Transactions on Programming Languages and Systems 13 (1) (1991) 124–149.

[7] M. Herlihy, N. Shavit, The asynchronous computability theorem for t-resilient tasks, in: Proceedings of the 25th ACM Symposium on Theory of Computing (STOC), 1993.

[8] M. Herlihy, J. M. Wing, Linearizability: a correctness condition for concurrent objects, ACM Transactions on Programming Languages and Systems 12 (3) (1990) 463–492.

[9] P. Jayanti, Wait-free computing, in: Proceedings of the 9th International Workshop on Distributed Algorithms (WDAG'95), vol. 972 of LNCS, Springer Verlag, 1995.

[10] P. Jayanti, Robust wait-free hierarchies, Journal of the ACM 44 (4) (1997) 592–614.

[11] M. C. Loui, H. H. Abu-Amara, Memory requirements for agreement among unreliable asynchronous processes, Advances in Computing Research (1987) 163–183.

[12] N. A. Lynch, Distributed Algorithms, Morgan Kaufmann Publishers, 1996.

[13] S. Moran, Y. Wolfstahl, Extended impossibility results for asynchronous complete networks, Inf. Process. Lett. 26 (3) (1987) 145–151.

[14] G. Neiger, Failure detectors and the wait-free hierarchy, in: Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC), 1995.

[15] E. Ruppert, Determining consensus numbers, SIAM Journal of Computing 30 (4) (2000) 1156–1168.