The Multiplicative Power of Consensus Numbers

Damien Imbs, Michel Raynal

Publications Internes de l'IRISA
ISSN : 2102-6327
PI 1943 – janvier 2010

The Multiplicative Power of Consensus Numbers

Damien Imbs[*] , Michel Raynal[**]
damien.imbs@irisa.fr, raynal@irisa.fr

Abstract: The Borowsky-Gafni (BG) simulation algorithm is a powerful reduction algorithm that shows that t-resilience of decision tasks can be fully characterized in terms of wait-freedom. Said in another way, the BG simulation shows that the crucial parameter is not the number n of processes but the upper bound t on the number of processes that are allowed to crash. The BG algorithm considers colorless decision tasks in the base read/write shared memory model. (Colorless means that if, a process decides a value, any other process is allowed to decide the very same value.)

This paper considers system models made up of n processes prone to up to t crashes, and where the processes communicate by accessing read/write atomic registers (as assumed by the BG) and (differently from the BG) objects with consensus number x (with $x \leq t < n$). Let $ASM(n,t,x)$ denote such a system model. While the BG simulation has shown that the models $ASM(n,t,1)$ and $ASM(t+1,t,1)$ are equivalent, this paper focuses the pair (t,x) of parameters of a system model. Its main result is the following: the system models $ASM(n_1,t_1,x_1)$ and $ASM(n_2,t_2,x_2)$ have the same computational power for colorless decision tasks if and only if $\lfloor \frac{t_1}{x_1} \rfloor = \lfloor \frac{t_2}{x_2} \rfloor$. As can be seen, this contribution complements and extends the BG simulation. It shows that consensus numbers have a multiplicative power with respect to failures, namely the system models $ASM(n,t',x)$ and $ASM(n,t,1)$ are equivalent for colorless decision tasks iff $(t \times x) \leq t' \leq (t \times x) + (x-1)$.

Key-words: Asynchronous processes, BG simulation, Consensus number, Distributed computability, Fault-Tolerance, Process crash failure, Reduction algorithm, t-Resilience, k-Set agreement, Shared memory system, Synchronization power, System model, Wait-freedom.

La puissance multiplicative des nombres de consensus

Résumé : *Ce rapport étudie la puissance multiplicative des nombres de consensus par rapport au nombre de fautes.*

Mots clés : *Processus asynchrones, Simulation BG, Nombre de consensus, Calculabilité distribuée, Tolérance aux défaillances, Défaillances par crash, Algorithme de réduction, t-Résilience, Mémoire partagée, Puissance de synchronisation, Sans-attente.*

---

[*] Projet ASAP: équipe commune avec l'INRIA, le CNRS, l'université Rennes 1 et l'INSA de Rennes
[**] Projet ASAP: équipe commune avec l'INRIA, le CNRS, l'université Rennes 1 et l'INSA de Rennes

1 Introduction

1.1 Context of the work

Wait-free implementation and consensus number [20] In the consensus problem, each process proposes a value and (a) every process that does not crash decides a value (termination), such that (b) a decided value is a proposed value (validity), and (c) no two processes decide different values (agreement). Enriching asynchronous read/write shared memory systems with consensus objects is fundamental as these objects make it possible to wait-free implement any concurrent object that has a sequential specification.

A *wait-free* implementation of a concurrent object ensures that every invocation of an object operation always terminates if the invoking process does not crash (i.e., whatever the behavior of the other processes, which can be very slow or even crashed). This means that, in shared memory systems made up of n processes, a *wait-free* implementation copes with up to $n-1$ process crashes. Hence, a fundamental question concerns the construction of wait-free consensus objects.

The *consensus number* notion captures the maximal synchronization power of an object with respect to the adversary net effect of failures and asynchrony. A concurrent object type μ has consensus number x, if x is the largest integer (or $+\infty$ if there is no such integer) such that a consensus object can be wait-free implemented from objects of type μ and shared read/write atomic registers in a system of x processes. As they allow any concurrent object (defined from a sequential specification) to be wait-free implemented in a system of x processes, the objects whose consensus number is greater than or equal to x are said to be *universal* in any system made up of at most x processes.

Shared read/write registers are the poorest synchronization objects as their consensus number is 1. Test&set objects are a little bit more powerful, as their consensus number is 2. Similarly, the consensus number of shared stacks or shared queues is 2. The consensus number of the n-register atomic write is $2n-2$. Differently, the consensus number of Compare&Swap objects is $+\infty$, which means that consensus can be solved for any number of processes, despite any number of crashes, from Compare&Swap objects and read/write registers. The consensus numbers establish what is called Herlihy's hierarchy of synchronization objects.

Borowsky-Gafni (BG) simulation [6] Let us consider an algorithm A that is assumed to solve a decision problem T (e.g., consensus) in an asynchronous read/write shared memory system made up of n processes, and where any subset of at most t processes may crash, i.e., A is t-resilient. Given A as input, the BG simulation is a powerful algorithm that solves T in an asynchronous read/write system made up of $t+1$ processes, where up to t may crash. Hence, the BG simulation is a wait-free algorithm.

The BG simulation has been used to prove solvability and unsolvability results in crash-prone read/write shared memory systems [8]. Basically, for a particular class of decision tasks called colorless tasks (those are the tasks where, if a process decides a value, any other process is allowed to decide the very same value), BG simulation characterizes t-resilience in terms of wait-freedom. (The BG simulation algorithm has been extended to colored tasks in [16, 23]). As an example, let us assume that A solves consensus, despite up to $t=1$ crash, among n processes in a read/write shared memory system. Taking A as input, the BG simulation builds a $(t+1)$-process (i.e., 2-process) algorithm A' that solves consensus despite $t=1$ crash. But, we know that consensus cannot be wait-free solved in a crash-prone asynchronous system made up of two processes that communicate by accessing shared read/write registers only [14, 20, 24]. It then follows that, whatever the number n of processes the system is made up of, there is no 1-resilient consensus algorithm.

Set consensus number [17] The k-set agreement problem has been introduced by Chaudhuri [11]. It is a weakened form of consensus, namely, the processes are allowed to decide up to k different proposed values. While it is trivial to solve k-set agreement in asynchronous read/write shared memory systems prone to up to $t<k$ process crashes, it has been shown that this becomes impossible when $t \geq k$ [6, 22, 28].

Gafni and Kuznetsov have introduced the notion of *set consensus number* of a decision task T. It is the greatest integer k such that T can be wait-free solved using read/write registers and k-set agreement objects. It follows that, in a system of n processes, the tasks can be categorized into n equivalence classes 1, 2, ..., n. The class 1 is the class of universal tasks (as it is equivalent to consensus, such a task can wait-free implement any other task in a system of n processes), while class n contains the trivial tasks that can be solved asynchronously in a crash-prone read/write shared memory system.

1.2 Content of the paper

Model parameters Let $ASM(n,t,x)$ denote a shared memory system model made up of n processes, in which up to t processes may crash ($1 \leq t < n$), and where the processes communicate and cooperate through shared read/write registers and objects with consensus number x ($1 \leq x \leq n$) (when $x > t$, all tasks can be solved). The parameter t measures the power of the *adversary* (that can crash arbitrarily up to t processes), while, on the "friend" side, x measures the *wait-free synchronization power* the processes can benefit from shared objects. Considering colorless tasks, the paper addresses the computability power of system models $ASM(n,t,x)$ such that $n > t$ and $n \geq x$.

Expressed with this notation, the BG simulation shows that $ASM(n,t,1)$ and $ASM(t+1,t,1)$ have the same power for decision tasks. Actually, assuming a pure asynchronous read/write system ($x=1$) and a given upper bound on the number of processes that

may crash (t), the BG simulation shows that the model parameter that is crucial is not the number n of processes, but t. Whatever the number of processes, t-resilience can be characterized by wait-freedom [6]. Hence, while the BG simulation addresses one face of the coin (namely, it considers $x = 1$ and addresses the equivalence between the pair (n, t) and the pair $(t + 1, t)$), this paper addresses the other face of the coin (namely, it considers a fixed number n of processes and addresses the equivalence between the pair (t', x) and the pair $(t, 1)$ when comparing $ASM(n, t', x)$ and $ASM(n, t, 1)$).

The question we are interested in As just indicated, this paper focuses on the system model parameters t and x. Let $ASM(n, t_1, x_1)$ and $ASM(n, t_2, x_2)$ be two system models such that $n > \max(t_1, t_2)$. The main question addressed in the paper is the following: "Which conditions the parameters t_1, x_1, t_2 and x_2 have to satisfy in order that any (colorless) task that can be solved (or is impossible to solve) in $ASM(n, t_1, x_1)$, can be solved (or is impossible to solve) in $ASM(n, t_2, x_2)$?" The paper answers this question by presenting two contributions.

Contribution #1 The first contribution is a necessary and sufficient condition that answers the previous question. This condition is $\lfloor \frac{t_1}{x_1} \rfloor \geq \lfloor \frac{t_2}{x_2} \rfloor$. This result has the following consequence: $ASM(n, t_1, x_1)$ and $ASM(n, t_2, x_2)$ are equivalent (have the same computational power) as far colorless decision tasks are concerned, if and only if $\lfloor \frac{t_1}{x_1} \rfloor = \lfloor \frac{t_2}{x_2} \rfloor$.

To that end, the paper presents two simulation algorithms. The first is a simulation of $ASM(n, t', x)$ into $ASM(n, t, 1)$. That simulation requires $t \leq \lfloor \frac{t'}{x} \rfloor$. The second is a simulation of $ASM(n, t, 1)$ into $ASM(n, t', x)$. It is shown that this simulation requires $t \geq \lfloor \frac{t'}{x} \rfloor$.This means that $ASM(n, t', x)$ and $ASM(n, t, 1)$ are equivalent iff $(t \times x) \leq t' \leq (t \times x) + (x - 1)$.

To illustrate the interest of these equivalences, let us consider the following simple examples that are immediate consequences of our results.

- Let us consider the system model $ASM(n, n-1, n-1)$, i.e., a system where processes are provided with objects with consensus number $n - 1$ (hence, consensus cannot be solved in this system of n processes). Moreover, as $t = n - 1$, an algorithm solving a task T in this system has to be wait-free.

  The paper shows that, for any colorless decision task T, (im)possibility results are the same in $ASM(n, n - 1, n - 1)$ and $ASM(n, 1, 1)$, and more generally in any system model $ASM(n, t, t)$. As, for any n, consensus cannot be solved in $ASM(n, n-1, n-1)$ (i.e., cannot be wait-free solved from objects with consensus number $n - 1$), it cannot be solved either in $ASM(n, 1, 1)$ (i.e., 1-resiliently in the base read/write model). This constitutes a new proof that, whatever the values of n and t, (a) consensus cannot be solved t-resiliently in $ASM(n, t, t)$ and, $\forall\ t' < t$, (b) the model $ASM(n, t', t)$ and the failure-free read/write model $ASM(n, 0, 1)$ are equivalent.

  More generally, these simulations provide us with a general proof of the impossibility of t-resilient tasks when processes can cooperate only through read/write registers and objects with consensus number t. A task that cannot be solved in $ASM(n, t, 1)$ cannot be solved either in $ASM(n, t', x)$ for the pairs (t', x) such that $t \leq \lfloor \frac{t'}{x} \rfloor$, and a task that can be solved in $ASM(n, t, 1)$ can be solved in $ASM(n, t', x)$ for pairs (t', x) such that $t \geq \lfloor \frac{t'}{x} \rfloor$.

- Let T_k be a task whose set consensus number is k. This task can be solved in $ASM(n, t, 1)$ for $t < k$ (because k-set agreement can be solved in $ASM(n, t, 1)$ when $t < k$), and cannot be solved in $ASM(n, t, 1)$ for $t \geq k$ (because k-set agreement cannot be solved in $ASM(n, t, 1)$ for $t \geq k$ [6, 22, 28]). Our result shows that T_k can be solved in any system $ASM(n, t', x)$ such that $\lfloor \frac{t'}{x} \rfloor \leq (k - 1)$, i.e., $t' \leq (k - 1) \times x + (x - 1) = k \times x - 1$ if x is fixed, or $x \geq \frac{t'+1}{k}$ if t' is fixed.

Contribution #2 As a side effect of the previous results, the second contribution of the paper is a generalization of the BG simulation. It shows that any task that can be solved (resp., is impossible to solve) in $ASM(n, t, x)$ can (resp., cannot) be solved in $ASM(t+1, t, x)$. (The case $x = 1$ does correspond to the BG simulation.)

A noteworthy feature of our results is that, following Gafni's reduction style, they are all obtained by algorithmic reductions. Hence, inspired by the BG simulation algorithm, the paper complements it by introducing another powerful simulation algorithm for proving possibility and impossibility results for decision tasks in asynchronous systems prone to process crashes. It also generalizes, unifies and extends previous results. As already said, the BG simulation and the proposed simulation are the two faces of the same coin.

1.3 Related works

The main works related to our work have been cited previously (namely, BG simulation [6], consensus number [20], and set consensus number [17]). Here we shortly describe a few additional results, related to distributed computability or efficiency, that have addressed issues that are in the same spirit as ours.

Results related to the dividing power of asynchrony Let A be an n-process algorithm designed for a round-based synchronous message-passing system prone to t process crashes. How many rounds of A is it possible to simulate in an asynchronous message-passing system (of n processes) prone to t' process crashes? It has been shown by Gafni that the first $\lfloor \frac{t}{t'} \rfloor$ rounds of A can be simulated [15]. This result exhibits a dividing power of asynchrony with respect to synchrony.

Using underlying base (m, ℓ)-set agreement objects An (m, ℓ)-set agreement object is an object that solves the ℓ-set agreement in a set of m processes. Given such objects, an interesting question concerns the wait-free implementation of an (n, k)-set agreement object from (wait-free) (m, ℓ)-set agreement objects, namely, which condition the values n, k, m, and ℓ have to satisfy for such an implementation to be possible in an asynchronous shared memory system. This question has first been posed by Borowsky and Gafni who have established a k-set agreement hierarchy [7] and showed that an (n, k)-set agreement object cannot be implemented from (m, ℓ)-set agreement objects when $\frac{n}{k} > \frac{m}{\ell}$. To that end, the authors present a simulation where the simulated processes access (m, ℓ)-set agreement objects, while the simulators access only snapshot objects. This simulation, that (as ours) is based on the BG simulation algorithm, has then been generalized in [12].

On the asynchronous decidability point of view, using topology-based arguments, Herlihy and Rajsbaum [21] have shown that it is possible to solve the k-set agreement problem when $k \geq \ell \lfloor \frac{t+1}{m} \rfloor + \min\big(\ell, (t+1) \mod m\big)$, and it is impossible to solve k'-set agreement for $k' < k$. On the efficiency point of view, Mostéfaoui, Raynal and Travers have shown [26] that, in synchronous message-passing systems enriched with (m, ℓ)-set agreement objects, the k-set agreement problem can be optimally solved in $\big\lfloor t/\big(m\lfloor \frac{k}{\ell} \rfloor + (k \mod \ell)\big)\big\rfloor + 1$ rounds.

Boosting the computability power with failure detectors Given the system model $ASM(n, n-1, x)$, an important question is the following: which is the weakest failure detector [10, 13] this system model has to be enriched with in order an object with consensus number $x+1$ can be built, i.e., in order to build $ASM(n, n-1, x+1)$? This is called a *boosting* problem. Guerraoui and Kuznetsov have shown that Ω_x is the weakest failure detector class for such a boosting [19].

Ω_x outputs, at each process, a set of x processes such that eventually the same set is output at all correct processes and this set contains at least one correct process [27]. Ω_x captures the exact information on failures that, when added to $ASM(n, n-1, x)$, is sufficient to build an object with consensus number $x+1$. This result generalizes in a precise sense the result of the weakest failure detector class to solve consensus in asynchronous systems (Ω_1 is Ω as defined in [10]).

1.4 Roadmap

The paper is made up of 6 sections. Section 2 introduces base definitions. Then, the two main simulations are presented. Section 3 presents a simulation of $ASM(n, t', x)$ into $ASM(n, t, 1)$ and shows that it requires $t \leq \lfloor \frac{t'}{x} \rfloor$. That simulation is an extension of the BG simulation that allows the simulated processes not only to write and snapshot a shared memory, but also to access objects with consensus number x. Then, Section 4 presents a simulation of $ASM(n, t, 1)$ into $ASM(n, t', x)$ and shows that it requires $t \geq \lfloor \frac{t'}{x} \rfloor$. This simulation is inspired by, but different from the BG simulation. Considering the system model parameter n, Section 5 generalizes the previous results. After having shown that $ASM(n, t', x)$ and $ASM(t+1, t, 1)$ have the same computational power if $t = \lfloor \frac{t'}{x} \rfloor$, it generalizes this result by showing that $ASM(n_1, t_1, x_1)$ and $ASM(n_2, t_2, x_2)$ have the same power if $\lfloor \frac{t_1}{x_1} \rfloor = \lfloor \frac{t_2}{x_2} \rfloor$. This section also discusses the multiplicative power of consensus numbers (with respect to the maximal number of process crashes). Finally, Section 6 concludes the paper.

2 Base definitions

2.1 Decision tasks

The problems we are interested in are called *decision tasks*[1]. In every run, each process proposes a value and the proposed values define an input vector I where $I[j]$ is the value proposed by p_j. Let $\mathcal{I}$ denote the set of allowed input vectors. Each process has to decide a value. The decided values define an output vector O, such that $O[j]$ is the value decided by p_j ($O[j]$ is locally kept by p_j in $output_j$). Let $\mathcal{O}$ be the set of the output vectors.

A decision task is a total binary relation Δ from $\mathcal{I}$ into $\mathcal{O}$. A task is *colorless* if, when a value v is decided by a process p_j (i.e., $O[j] = v$), then the very same value v can be decided by any other process (k-set agreement is a colorless task). Hence, $\mathcal{O}$ contains vectors whose all values are equal. Consensus, and more generally k-set agreement, are colorless tasks. Otherwise the task is *colored*. Renaming [3, 9, 22] is a colored task.

[1] The reader interested in a more formal presentation of decision tasks can consult the literature (e.g., [2, 8, 16, 22, 23]).

2.2 System model: $ASM(n,t,x)$

The asynchronous shared memory model $ASM(n,t,x)$ ($1 \leq t < n$) has been briefly presented in the introduction. This section describes it more precisely. (A formal definition of a system model can be found in [8].)

Asynchronous processes and fault model As already indicated, we are interested in distributed algorithms the aim of which is to solve a task in a system made up of n asynchronous sequential processes denoted p_1, ..., p_n. A process executes a sequence of atomic steps (as defined by its algorithm).

A process can crash in a run. A process executes correctly the steps defined by its algorithm until it crashes (if it ever does). After if has crashed, a process executes no more steps. If it does not crash, a process executes an infinite number of steps.

It is assumed that an arbitrary subset (not known in advance) of up to $t < n$ processes can crash (the crash of one process being independent from the crash of other processes). A process that does not crash in a run is said to be *correct* in that run, otherwise it is *faulty*. This failure model is called the *t-resilient environment*, and an algorithm designed for such an environment is said to be *t-resilient*. The extreme case $t = n-1$ is called *wait-free environment*, and the corresponding algorithms are called *wait-free algorithms*.

Communication model The processes communicate by accessing a shared read/write memory, and (if $x > 1$) objects whose consensus number is x.

- The shared read/write memory is a snapshot object [1] denoted $mem[1..n]$, that has one entry $mem[j]$ per process p_j. The process p_j is the only one that can write $mem[j]$. It does it by invoking $mem[j].\mathsf{write}(v)$ where v is the new value of $mem[j]$. Initially, all entries of mem contains the default value $\perp$. Any process p_j can atomically read the array $mem[1..n]$ by invoking $mem.\mathsf{snapshot}()$. Let us remember that such a snapshot object can be wait-free implemented on top of atomic read/write registers [1, 4].
- If $x > 1$, the processes can access as many consensus objects with consensus number x as they want, but a given object cannot be accessed by more than x (statically defined) processes. Using an array-like notation, such an object is denoted $x\_cons[a]$. A process p_i, allowed to access $x\_cons[a]$, accesses it by invoking $x\_cons[a].\mathsf{x\_cons\_propose}(v)$ where v is the value it proposes to that consensus object. If p_i is correct, that invocation returns it the value decided from $x\_cons[a]$.

2.3 Algorithm solving a task

An algorithm solves a task in a t-resilient environment if, given any $I \in \mathcal{I}$, each correct process p_j decides (i.e., writes a value v in $output_j$) and there is an output vector O such that $(I,O) \in \Delta$ where O is defined as follows. If p_j decides v, then $\mathcal{O}[j] = v$. If p_j does not decide, $O[j]$ is set to any value v' that preserves the relation $(I,O) \in \Delta$.

A task is solvable in a t-resilient environment if there is an algorithm that solves it in that environment. As an example, consensus is not solvable in the 1-resilient environment [14, 24, 25]. Differently, renaming with $2n-1$ names is solvable in the wait-free environment [3, 5, 22].

2.4 Simulated processes vs simulator processes

Aim Let A be an algorithm that solves a decision task in the $ASM(n,t,x)$ model as described previously (hence, A is t-resilient). The aim is to design an algorithm A' that simulates A in $ASM(n,t',x')$ (hence, A is t'-resilient). The reader is referred to [8] for a formal definition of a simulation of a system model into another system model.

Notation A simulated process (that executes in a "source" system model $ASM(n,t,x)$) is denoted p_j with $1 \leq j \leq n$, and the subscript j is always used to refer to a simulated process. Similarly, a simulator process (that executes in a "target" system model $ASM(n,t',x')$) is denoted q_i with $1 \leq i \leq n$, and the subscript i is always used to refer to a simulator.

As far as the objects accessed by the simulators are concerned, the following convention is adopted. The objects denoted with upper case letters are the objects shared by the simulators. Conversely, an object denoted with lower case letters is local to a simulator (in that case, the associated subscript denotes the corresponding simulator).

What a simulator does Each simulator q_i is given the code of every simulated process $p_1, \ldots, p_n$. It manages n threads, each one associated with a simulated process, and locally executes these threads in a fair way. It also manages a local copy mem_i of the snapshot memory mem shared by the simulated processes, and a local copy $x\_cons[a]_i$ of each consensus number x object shared by these processes.

The code of a simulated process p_j contains invocations of $mem[j].\mathsf{write}()$, $mem.\mathsf{snapshot}()$, and $x\_cons[a].\mathsf{x\_cons\_propose}()$ (if p_j is one of the x processes that access $x\_cons[a]$). These are the only operations used by the processes $p_1, \ldots, p_n$ to cooperate. So, the core of the simulation is the design of algorithms that describe how a simulator q_i simulates these operations (within $ASM(n,t',x')$) when

invoked by a simulated process p_j (within $ASM(n,t,x)$). These simulation algorithms are denoted $\mathsf{sim\_write}_{i,j}()$, $\mathsf{sim\_snapshot}_{i,j}()$ and $\mathsf{sim\_x\_cons\_propose}_{i,j}()$.

3 Simulating $ASM(n,t',x)$ in $ASM(n,t,1)$

Let A be an algorithm that is assumed to solve a task T in $ASM(n,t',x)$. This means that A is a t'-resilient algorithm that can use objects with consensus number x. This section presents an algorithm that, given A as input, produces a (t-resilient) algorithm A' that solves T in $ASM(n,t,1)$. Assuming $\max(t,t') < n$, this simulation requires $t \leq \lfloor \frac{t'}{x} \rfloor$.

The BG simulation provides us with appropriate implementations of the operations $mem[j].\mathsf{write}()$ and $mem.\mathsf{snapshot}()$ when we want to simulate, into $ASM(t+1,t,1)$, a colorless decision task designed for $ASM(n,t,1)$. As we are about to see, the proposed algorithm that simulates, into $ASM(n,t,1)$, an algorithm A designed for $ASM(n,t',x)$, is actually a simple extension of the BG simulation. It borrows the operations $mem[j].\mathsf{write}()$ and $mem.\mathsf{snapshot}()$ from the BG simulation, and provides an additional implementation for the $\mathsf{x\_cons\_propose}()$ operation on a consensus number x object.

Remark The simulations presented in [7] and [12] allow the simulated processes to access objects stronger than atomic read/write registers, namely, (m,ℓ)-set consensus objects, while the simulators are restricted to access snapshot objects only. To that end, these simulations direct the simulators to agree twice. The simulators have first to agree on at most ℓ values (to that end they use a new $(\ell-1)$-resilient object built on top of snapshot objects). They have then to agree in order that each simulated process obtains the same value whatever the simulator.

These simulations could be customized in order to simulate $ASM(n,t',x)$ in $ASM(n,t,1)$. The simulation that follows is simpler. In order to simulate a consensus object, it only needs a single safe_agreement type object (that is the same type as the one used in the original BG simulation, see below).

3.1 The safe_agreement object type

The safe_agreement type This object type (defined in [6, 8]) is at the core of the BG simulation. It provides each simulator q_i with two operations, denoted $\mathsf{sa\_propose}(v)$ and $\mathsf{sa\_decide}()$, that q_i can invoke at most once, and in that order. The operation $\mathsf{sa\_propose}(v)$ allows q_i to propose a value v while $\mathsf{sa\_decide}()$ allows it to decide a value. The properties satisfied by an object of the type safe_agreement are the following.

- Termination. If no simulator crashes while executing $\mathsf{sa\_propose}()$, then any correct simulator that invokes $\mathsf{sa\_decide}()$ returns from that invocation.
- Agreement. At most one value is decided.
- Validity. A decided value is a proposed value.

init: **for each** $x : 1 \leq x \leq n$ **do** $SM[x] \leftarrow (\perp, 0)$ **end for**.

operation $\mathsf{sa\_propose}_i(v)$:
(01) $SM[i] \leftarrow (v,1)$;
(02) $sm_i \leftarrow SM.\mathsf{snapshot}()$;
(03) **if** $(\exists x : sm_i[x].level = 2)$ **then** $SM[i] \leftarrow (v,0)$ **else** $SM[i] \leftarrow (v,2)$ **end if**.

operation $\mathsf{sa\_decide}_i()$:
(04) **repeat** $sm_i \leftarrow SM.\mathsf{snapshot}()$ **until** $(\forall x : sm_i[x].level \neq 1)$ **end repeat**;
(05) **let** $x = \min(\{k \mid sm_i[k].level = 2)$; $res \leftarrow sm_i[x].value$;
(06) $\mathsf{return}(res)$.

Figure 1: An implementation of the safe_agreement type [8] (code for simulator q_i)

An implementation The implementation of the safe_agreement type described in Figure 1 is from [8]. This construction is based on a snapshot object SM (with one entry per simulator q_i). Each entry $SM[i]$ of the snapshot object has two fields: $SM[i].value$ that contains a value and $SM[i].level$ that stores its level. The level 0 means the corresponding value is meaningless, 1 means it is unstable, while 2 means it is stable.

When a simulator q_i invokes $\mathsf{sa\_propose}_i(v)$, it first writes the pair $(v,1)$ in $SM[i]$ (line 01), and then reads the snapshot object SM (line 02). Then (line 03), if there is a stable value in SM, p_i "cancels" the value it proposes (its state becomes "meaningless"), otherwise it makes it stable (its state becomes "stable").

A simulator q_i invokes sa_decide() after it has invoked sa_propose(). Its aim is to return the same stable value to all the simulators that invoke this operation (line 06). To that end, q_i repeatedly computes a snapshot of SM until it sees no unstable value in SM (line 04). Let us observe that, as a simulator q_i invokes sa_decide() after it has invoked sa_propose(v), there is at least one stable value in SM when it executes line 05. Finally, in order that the same stable value be returned to all, q_i returns the stable value proposed by the simulator with the smallest id (line 05). A formal proof showing that this algorithm implements the safe_agreement type can be found in [8].

3.2 Simulating $mem[j]$.write() and mem.snapshot()

The simulation by a simulator q_i of these operations issued by a simulated process p_i is the same as in the BG simulation. They are described here (using our notation) to make the paper self-contained.

3.2.1 The shared memory $MEM[1..n]$

The snapshot memory mem shared by the simulated processes $p_1, \ldots, p_n$ is emulated by a snapshot object $MEM[1..n]$ shared by the simulators $q_1, \ldots, q_n$. More specifically, $MEM[i]$ is an atomic register that contains an array with one entry per simulated process p_j. Each $MEM[i][j]$ is made up of two fields: a field $MEM[i][j].value$ that contains the last value of $mem[j]$ written by p_j, and a field $MEM[i][j].sn$ that contains the associated sequence number. (This sequence number, introduced by the simulation, is a control data that will be used to produce a consistent simulation of the mem.snapshot() operations issued by the simulated processes p_j).

3.2.2 The sim_write$_{i,j}$() operation

The algorithm, denoted sim_write$_{i,j}(v)$, executed by q_i to simulate the write by p_j of the value v into $mem[j]$ is described in Figure 2 [8]. Its code is pretty simple. The simulator q_i first increases a local sequence number $w\_sn_i[j]$ that is associated with the value v written by p_j into $mem[j]$. Then, q_i writes the pair $(v, w\_sn_i[j])$ into $mem_i[j]$ (where mem_i is its local copy of the memory shared by the simulated processes) and finally writes atomically its local copy mem_i into $MEM[i]$.

operation sim_write$_{i,j}(v)$:
(01) $w\_sn_i[j] \leftarrow w\_sn_i[j] + 1$;
(02) $mem_i[j] \leftarrow (v, w\_sn_i[j])$;
(03) $MEM[i] \leftarrow mem_i$.

Figure 2: sim_write$_{i,j}(v)$ executed by q_i to simulate $mem[j]$.write(v) issued by p_j (from [8])

3.2.3 The sim_snapshot$_{i,j}$() operation

This operation is implemented by the algorithm described in Figure 3 [8].

Additional local and shared objects For each process p_j, a simulator q_i manages a local sequence number generator $snap\_sn_i[j]$ used to associate a sequence number with each mem.snapshot() it simulates on behalf of p_j (line 04).

In addition to the snapshot object $MEM[1..n]$, the simulators $q_1, \ldots, q_n$ cooperate through an array $SAFE\_AG[1..n, 0.. + \infty)$ of safe_agreement type objects.

Underlying principle of the BG simulation [6, 8]: obtaining a consistent value In order to agree on the very same output of the $snapsn$-th invocation of mem.snapshot() that is issued by p_j, the simulators $q_1, \ldots, q_n$ use the object $SAFE\_AG[j, snapsn]$.

Each simulator q_i proposes a value (denoted $input_i$) to that object (line 05) and, due to its agreement property, that object will deliver them the same output at line 06. In order to ensure the consistent progress of the simulation, the input value $input_i$ proposed by the simulator q_i to $SAFE\_AG[j, snapsn]$ is defined as follows.

- First, q_i issues a snapshot of MEM in order to obtain a consistent view of the simulation state. The value of this snapshot is kept in sm_i (line 01).

  Let us observe that $sm_i[x][y]$ is such that (1) $sm_i[x][y].sn$ is the number of writes issues by p_y into $mem[y]$ that have been simulated up to now by q_x, and (2) $sm_i[x][y].value$ is the value of the last write into $mem[y]$ as simulated by q_x on behalf of p_y.

- Then, for each p_y, q_i computes $input_i[y]$. To that end, it extracts from $sm_i[1..n][y]$ the value written by the more advanced simulator q_s as far as the simulation of p_y is concerned. This is expressed in lines 02-03.

```
operation sim_snapshot_{i,j}():
(01) sm_i ← MEM.snapshot():
(02) for each y : 1 ≤ y ≤ n: do input_i[y] = sm_i[s][y].value
(03)          where ∀x : 1 ≤ x ≤ n : sm_i[s][y].sn ≥ sm_i[x][y].sn end for;
(04) snap_sn_i[j] ← snap_sn_i[j] + 1; let snapsn = snap_sn_i[j];
(05) enter_mutex1; SAFE_AG[j, snapsn].sa_propose_i(input_i); exit_mutex1;
(06) res ← SAFE_AG[j, snapsn].sa_decide_i()
(07) return(res).
```

Figure 3: sim_snapshot$_{i,j}()$ executed by q_i to simulate mem.snapshot() issued by p_j (from [8])

Once, $input_i$ has been computed, q_i proposes it to $SAFE\_AG[j, snapsn]$ (line 05), and then returns the value decided by that object (lines 06-07).

The previous description shows an important feature of the BG simulation. A value $input_i[y] = sm_i[s][y].value$ proposed by a simulator q_i can be such that $sm_i[s][y].sn > sm_i[i][y].sn$, i.e., the simulator q_s is more advanced than q_i as far as the simulation of p_y is concerned. This causes no problem, as when q_i will simulate mem.snapshot() operations for p_y (if any) that are between the $(sm_i[i][y].sn)$-th and the $(sm_i[s][y].sn)$-th write operations of p_y, it will obtain a value that has already been computed and is currently kept in the corresponding $SAFE\_AG[y, -]$ object.

Underlying principle of the BG simulation [6, 8]: a key issue Each simulator q_i simulates the n processes $p_1, \ldots, p_n$ "in parallel" and in a fair way. But any simulator q_i can crash. The crash of q_i while it is engaged in the simulation of mem.snapshot() on behalf of several processes p_j, $p_{j'}$, etc., can entail their definitive blocking, i.e., their crash. This is because each $SAFE\_AG[j, -]$ object guarantees that its $SAFE\_AG[j, -]$.decide() invocations do terminate only if no simulator crashes while executing $SAFE\_AG[j, -]$.propose() (line 05 of Figure 3).

The simple (and bright) idea of the BG simulation to solve this problem consists in allowing a simulator to be engaged in only one $SAFE\_AG[-, -]$.propose() invocation at a time. Hence, if q_i crashes while executing $SAFE\_AG[j, -]$.propose(), it can entail the crash of p_j only. This is obtained by using an additional mutual exclusion object offering the operations enter_mutex1 and exit_mutex1. (Let us notice that such a mutex object is purely local to each simulator: it solves conflicts among the simulating threads inside each simulator, and has nothing to do with the memory shared by the simulators).

3.3 Simulating $x\_cons[a]$.x_cons_propose()

As indicated, the simulated processes p_j cooperate by writing and snapshoting a shared memory mem and accessing objects with consensus number x. Let $x\_cons[a]$ be such an object. Let us remember that such an object is a one-shot object (a process invokes $x\_cons[a]$.x_cons_propose(v) at most once).

The implementation by a simulator q_i of an invocation $x\_cons[a]$.x_cons_propose(v) (issued by one of the x processes p_j that can access $x\_cons[a]$) is described in Figure 4. The value decided from $x\_cons[a]$ is computed by the simulators with the help of a safe_agreement object denoted $XSAFE\_AG[a]$. It is then saved by a simulator q_i in a local variable denoted $xres_i[a]$ (the initial value of which is $\perp$).

More precisely, if that value is already known by q_i when it invokes $x\_cons[a]$.x_cons_propose(v) on behalf of p_j (this occurs when q_i has already invoked $x\_cons[a]$.x_cons_propose() on behalf of another process $p_{j'}$), q_i returns it. Otherwise, q_i proposes v to the safe_agreement object $XSAFE\_AG[a]$, and then returns the value decided from that object. Moreover, as $XSAFE\_AG[a]$ is a one-shot object, a simulator q_i has to invoke at most once $XSAFE\_AG[a]$.sa_propose$_i()$ and $XSAFE\_AG[a]$.sa_decide$_i()$ (in that order). In order to prevent q_i from invoking these operations more than once, the access to the local variable $xres_i[a]$ is protected by a mutual exclusion mechanism (local to q_i and implemented by enter_mutex2 and exit_mutex2).

```
operation sim_x_cons_propose^a_{i,j}(v):
(01) enter_mutex2;
(02) if (xres_i[a] = ⊥) then enter_mutex1; XSAFE_AG[a].sa_propose_i(v); exit_mutex1;
(03)                          xres_i[a] ← XSAFE_AG[a].sa_decide_i()
(04) end if;
(05) exit_mutex2;
(06) return(xres_i[a]).
```

Figure 4: sim_x_cons_propose$^a_{i,j}()$ executed by q_i to simulate $x\_cons[a]$.x_cons_propose() issued by p_j

If the simulator q_i crashes while executing $XSAFE\_AG[a]$.sa_propose(v), it can block forever the simulation of the x processes that access the simulated object $x\_cons[a]$. On another side, the mutual exclusion (local to q_i) realized by enter_mutex2 and exit_mutex2

guarantees that a simulator can simulate at most one invocation $x\_cons[]$.x_cons_propose() at a time. It follows that, as far as the simulation of x_cons_propose() are concerned, if τ simulators crash, they can entail the crash of $\tau \times x$ simulated processes.

On another side, the crash of q_i while it has concurrently issued $XSAFE\_AG[a]$.sa_propose$_i$() (on behalf of a simulated process p_j) and $SAFE\_AG[j', snapsn]$.sa_propose$_i$() (on behalf of a simulated process $p_{j'}$) could entail the crash of $x+1$ processes. In order to prevent this from occurring, a simulator q_i is allowed to invoke at most one sa_propose$_i$() at a time. This is realized by calling again enter_mutex1 and exit_mutex1 before and after $XSAFE\_AG[a]$.sa_propose$_i$(), respectively. It follows that the crash of a simulator entails at most either the crash of a single simulated process (if q_i crashes during an invocation of $SAFE\_AG[j, snapsn]$.sa_propose$_i$()) or the crash of at most x processes (if q_i crashes during an invocation of $XSAFE\_AG[a]$.sa_propose$_i$()). As up to t simulators may crash, that is why the simulation requires $t \leq \lfloor \frac{t'}{x} \rfloor$. (It is important to see that enter_mutex2 is invoked before checking the value of $xres_i[a]$. This is required in order that a simulator invokes $XSAFE\_AG[a]$.sa_propose$_i$() at most once, as demanded by the specification of the one-shot safe_agreement object type.)[2]

3.4 Simulating $ASM(n, t', x)$ in $ASM(n, t, 1)$: correctness proof

What has to be proved: on the liveness side In order to always terminate, a t'-resilient algorithm requires that at most t' (simulated) processes crash. We show here that, when at most $t \leq \lfloor \frac{t'}{x} \rfloor$ simulators can crash (i.e. $t' \geq t \times x$), any correct simulator q_i can execute, without being blocked forever on any of these codes, the code of at least $(n - t')$ simulated processes p_j that access snapshot objects and objects with consensus number x.

What has to be proved: on the safety side On that side, we have the following.

- It has to be proved that a simulated process p_j that executes mem.snapshot() obtains the same value at each simulator q_i. Moreover, the snapshot values that are returned have to be consistent with the write operations $mem[j']$.write() issued by the simulated processes $p_{j'}$.
- In addition to snapshot objects, the simulated processes p_j can access objects $xcons[a]$ with consensus number x (operation x_cons_propose()). It has to be proved that, for every object $xcons[a]$, the x simulated processes that access it, obtain the same decided value.

Lemma 1 *The crash of a simulator q_i can entail the permanent blocking of at most x simulated processes p_j.*

Proof Let us observe that the crash of a simulator q_i can block the simulation of a simulated process p_j only when q_i accesses an underlying safe_agreement type object. This can occur only when q_i executes a sim_snapshot() operation (to simulate mem.snapshot() issued by p_j), or a sim_x_cons_propose() operation (to simulate an invocation of x_cons_propose() on an object $xcons[a]$).

If q_i crashes while executing sim_snapshot(), due to mutex mechanism (mutex1), it can definitely block only p_j. If q_i crashes while executing sim_x_cons_propose(), due to mutex mechanism (mutex2), it can definitely block only the x processes that access $xcons[a]$. Moreover, as (due to mutex1) q_i is constrained to access at most one safe_agreement type object, it follows that its crash can entail the definitive blocking of one or x simulated processes. $\square_{Lemma\ 1}$

Let us remember that the simulated processes $p_1, \ldots, p_n$ solve a decision task. This means that, in $ASM(n, t', x)$, each p_j that does not crash does *decide* a value.

Lemma 2 *Let $t \leq \lfloor \frac{t'}{x} \rfloor$. Each correct simulator q_i computes the decision value of at least $(n - t')$ simulated processes p_j.*

Proof Because (a) at most t simulators may crash, (b) a simulator can block at most x simulated processes (Lemma 1), and (c) $t' \geq t \times x$, it follows that each simulator q_i can execute the code of at least $n - t \times x \geq (n - t')$ simulated processes until they decide. Because the algorithm executed by the simulated processes $p_1, \ldots, p_n$ is t'-resilient, these $n - t \times x$ simulated processes eventually decide a value. Consequently, each simulator computes the decision values of at least $(n - t')$ simulated processes. $\square_{Lemma\ 2}$

Lemma 3 *The simulators that return from the simulation of the k-th invocation of mem.snapshot() issued by a simulated process p_j, obtain the same value for that simulated snapshot invocation.*

Proof The k-th invocation of mem.snapshot() issued by a simulated process p_j, is simulated by a corresponding sim_snapshot() invocation issued by each simulator q_i. Due to the agreement property of the underlying safe_agreement objects used to implement sim_snapshot(), it follows that the same value is returned to any simulator (lines 05-06 of Figure 3). $\square_{Lemma\ 3}$

[2] It is possible to replace enter_mutex2 and exit_mutex2 by enter_mutex1 and exit_mutex1 in Figure 4, and suppress enter_mutex1 and exit_mutex1 at line 02. We have not done it, in order to associate a single meaning with each mutex mechanism.

Lemma 4 *Let $xcons[a]$ be a consensus number x object accessed by a set of x simulated processes. All the simulators that invoke the operation $\mathsf{sim\_x\_cons\_propose}^a_{i,j}()$ associated with the invocations of the operation $xcons[a].\mathsf{x\_cons\_propose}()$ issued by the (allowed) x processes, obtain the same value.*

Proof The proof is similar to the one of the previous lemma. The underlying safe_agreement object $XSAFE\_AG[a]$, used to implement $xcons[a]$, ensures that at most one value can be decided (lines 02-03 of Figure 4). $\square_{Lemma\ 4}$

Lemma 5 *For every simulated process p_j, the simulators decide at most one value.*

Proof Let us observe that the $mem.\mathsf{snapshot}()$ operations, and (for any a) the $xcons[a].\mathsf{x\_cons\_propose}()$ operations are the only non-deterministic operations that a simulated process can issue. It follows from Lemma 3 and Lemma 4 that all simulators q_i obtain the same value when they invoke $\mathsf{sim\_snapshot}_{i,j}()$ or $\mathsf{sim\_x\_cons\_propose}^a_{i,j}()$ (that simulates $mem.\mathsf{snapshot}()$ and $xcons[a].\mathsf{x\_cons\_propose}()$ issued by p_j), which proves the lemma. $\square_{Lemma\ 5}$

Lemma 6 *The sequences of $\mathsf{sim\_write}_{i,j}()$, $\mathsf{sim\_snapshot}_{i,j}()$ and $\mathsf{sim\_x\_cons\_propose}^a_{i,j}()$ issued by each simulator (in $ASM(n,t,1)$) on behalf of each simulated processes p_j define a correct execution of the simulated algorithm.*

Proof Let us first observe that, due to Lemma 3 and Lemma 4, all the simulator q_i that are not blocked while simulating a process p_j, obtain the same value when p_j executes a non-deterministic statement issued by p_j. It follows that they all simulate p_j in the same way.

When a simulator q_i executes $\mathsf{sim\_snapshot}_{i,j}()$ (i.e. when it simulates an invocation of the multi-shot operation $mem.\mathsf{snapshot}()$ issued by p_j), it stores in its local variable $input_i$ the values written by the simulators that have advanced the most for each simulated process (see Figure 2, and lines 02-03 of Figure 3). Let us remember that, due to Lemma 3, once it has been assigned a value, an safe_agreement keeps that value forever. Thus, for each $\mathsf{sim\_snapshot}_{i,j}()$, q_i returns an $input$ value computed either by itself or another simulator. Let us notice that, at the time at which this $input$ value is determined, no simulator q_x has yet terminated its associated $\mathsf{sim\_snapshot}_{x,j}()$ (if it was the case, the value decided from the corresponding safe_agreement object would be the value $input$ proposed by q_i). Because each process p_j is simulated in a deterministic way, the $input$ value returned contains the last value written by p_j as seen by q_i. This shows that the process order of each simulated process p_j is respected.

To ensure that the simulation is correct, it remains to show that the $mem.\mathsf{write}()$, $mem.\mathsf{snapshot}()$, and $x\_cons[a].\mathsf{x\_cons\_propose}()$ operations issued by every simulated process p_j can be linearized. To that end, the linearization points are defined as follows.

- The linearization point of a $mem[j].\mathsf{write}()$ issued by p_j is placed at line 03 of Figure 2 of the first simulator q_i that executes $\mathsf{sim\_write}_{i,j}()$.
- The linearization point of $mem.\mathsf{snapshot}()$ issued by p_j is placed at line 01 of Figure 3 of the simulator q_i that imposes its $input_i$ value to the corresponding safe_agreement object.
- The linearization point of the one-shot operation $x\_cons[a].\mathsf{x\_cons\_propose}()$ issued by p_j is placed at line 02 of Figure 4 of the simulator q_i whose invocation $XSAFE\_AG[a].\mathsf{sa\_propose}_i()$ imposes its value to the corresponding safe_agreement object $XSAFE\_AG[a]$.

Because the simulator q_i that imposes its $input_i$ value for a $\mathsf{sim\_snapshot}()$ operation reads the most advanced values at the time of its snapshot (lines 02-03 of Figure 3), and because (due to Lemma 3) once a simulator finishes the execution of $\mathsf{sim\_snapshot}()$, the value for this $\mathsf{sim\_snapshot}()$ is fixed forever, the linearization corresponds to a linearization of a correct execution of the simulated algorithm as far as the $mem[j].\mathsf{write}()$ and $mem.\mathsf{snapshot}()$ operations are concerned. A similar reasoning shows that the linearization is also correct with respect to the $x\_cons[a].\mathsf{x\_cons\_propose}()$ operations issued by the simulated processes p_j. $\square_{Lemma\ 6}$

Theorem 1 *Let $t \leq \lfloor \frac{t'}{x} \rfloor$. The algorithms described in Figure 2, Figure 3, and Figure 4 are a correct simulation of $ASM(n,t',x)$ into $ASM(n,t,1)$ (for any algorithm A solving a colorless decision task in $ASM(n,t',x)$).*

Proof The proof that the simulation is correct follows directly from Lemma 2 for its liveness, and Lemma 5 and Lemma 6 for its safety. $\square_{Theorem\ 1}$

4 Simulating $ASM(n,t,1)$ in $ASM(n,t',x)$

4.1 The objects shared by the simulators

Let A be an algorithm that solves a colorless decision task in $ASM(n,t,1)$. This means that A is an n-process t-resilient algorithm and its processes p_1, ..., p_n cooperate by accessing a snapshot shared memory $mem[1..n]$ with $mem[j].\mathsf{write}()$ and $mem.\mathsf{snapshot}()$ operations.

Assuming $t \geq \lfloor \frac{t'}{x} \rfloor$, this section describes a simulation of A in the system model $ASM(n, t', x)$, where a simulator q_i can additionally access objects with consensus number x. The simulators access a snapshot memory $MEM[1..n]$ as in the previous simulation. The simulation of $mem[j]$.write() is exactly the same as the one described by the operation sim_write$_{i,j}()$ defined in Figure 2.

The main issue is the design of the operation sim_snapshot$_{i,j}()$ executed by q_i to simulate the invocation of mem.snapshot() by p_j. The difficulty comes from the fact that up to t' simulators are allowed to crash in $ASM(n, t', x)$, while only t simulated processes are allowed to crash in $ASM(n, t, 1)$, and t' can be greater than t.

4.2 The x_safe_agreement object type

Type definition This object type is an extension of the safe_agreement type described in Section 3.1. It is defined by two operations, denoted x_sa_propose() and x_sa_decide() that a simulator may invoke (at most once and in that order). Moreover, every x_safe_agreement object considers that x of the simulators are its *owners* (as we will see, different objects do not necessarily have the same set of owners). The properties defining the object type x_safe_agreement are the following.

- Termination. If at most $(x-1)$ owners crash while executing x_sa_propose(), then any correct simulator that invokes x_sa_decide(), returns from that invocation.
- Agreement. At most one value is decided.
- Validity. A decided value is a proposed value.

As we will see, while the case $x = 1$ boils down to the definition of the safe_agreement object type, the implementation of the x_safe_agreement type is not a straightforward extension of the algorithm described in Figure 1.

The sim_snapshot$_{i,j}$ operation Let us consider Section 3.2.3 where, instead of safe_agreement objects, the array $SAFE\_AG[1..n, 1..+\infty)$ contains now x_safe_agreement objects. The algorithm sim_snapshot$_{i,j}()$ invoked by a simulator q_i to simulate the invocation mem.snapshot() issued by a simulated process p_j, is then the same as the one described in Figure 3 after having replaced sa_propose() and sa_decide() by x_sa_propose() and x_sa_decide(), respectively.

4.3 Implementing the x_safe_agreement type

The implementation of a x_safe_agreement object is at the core of the simulation. It relies on consensus number x objects (the corresponding access operation is denoted x_cons_propose()), multi-writer/multi-reader atomic registers (let us remember that these registers can be implemented on top of a snapshot shared memory), and one-shot test&set objects (that can be implemented from consensus number x objects [18]).

Dynamically associating owners with an x_safe_agreement object Let us remember that, due to the operations enter_mutex1 and exit_mutex1 that are invoked before and after x_sa_propose$_i(v)$, respectively (see Figure 3), a simulator q_i is engaged in at most one x_safe_agreement object at a time.

On another side, as specified in its termination property, an x_safe_agreement object can "crash" when its x owners crash ("crash" of an x_safe_agreement object means that the simulated processes that invoke x_sa_decide$_i()$ on that object remain blocked forever). This means that if all the x_safe_agreement objects had the same set of x owners, constraining a simulator to be engaged in a single x_safe_agreement object at a time is not sufficient because, whenever they occur, their crashes would crash all the x_safe_agreement objects and the simulation could block forever. To prevent this from occurring, the simulation imposes that the owners of an object are determined dynamically. Intuitively, they are the "first" x processes that invoke x_sa_propose$_i()$ on that object. It follows that if t' simulators crash, they entail the crash of at most $\lfloor \frac{t'}{x} \rfloor$ x_safe_agreement objects, which in turn can block forever at most $\lfloor \frac{t'}{x} \rfloor$ simulated processes. Hence, the constraint $t \geq \lfloor \frac{t'}{x} \rfloor$.

```
operation x_compete_i():
(01) ℓ ← 1; winner ← false;
(02) while (ℓ ≤ x ∧ ¬winner) do
(03)       winner ← TS[ℓ].test&set(); ℓ ← ℓ + 1
(04) end while;
(05) return(winner).
```

Figure 5: An implementation of the x_compete$_i()$ operation (code for q_i)

An object denoted $X\_T\&S$ is associated with each x_safe_agreement object. It provides the simulators with a single operation, denoted x_compete$_i()$ that returns $true$ to x simulators (if x or less processes invoke it, the ones that do not crash all obtain $true$). This

operation is implemented by the algorithm described in Figure 5 that uses an array of x test&set objects. Such an object returns $true$ to the first invocation, and $false$ to the following invocations. As its consensus number is 2, a test&set object can easily be implemented from an object with consensus number x.

The simulators that obtain $true$ when they invoke $X\_T\&S.\mathsf{x\_compete}_i()$ are the owners of the corresponding x_safe_agreement object. It is easy to see that, as a simulator can invoke $\mathsf{x\_sa\_propose}_i()$ on at most one x_safe_agreement object at a time, and as a simulator can be owner only of the object for which it has a pending $\mathsf{x\_sa\_propose}_i()$ invocation, no simulator can be the owner of several x_safe_agreement objects at the same time.

The operation $\mathsf{x\_sa\_decide}_i()$ The value decided from a x_safe_agreement object is saved in an associated atomic register X_SAFE_AG (initialized to $\perp$). Consequently, when a simulator q_i invokes $\mathsf{x\_sa\_decide}_i()$ on that x_safe_agreement object, it first waits until that register has been assigned a value, and then returns that value (See lines 09-10 in Figure 6).

The operation $\mathsf{x\_sa\_propose}_i(v)$ The algorithm implementing $\mathsf{x\_sa\_propose}_i(v)$ for a given object of type x_safe_agreement is described in Figure 6. A simulator q_i first invokes $X\_T\&S.\mathsf{x\_compete}_i()$ to know if it is an owner of that object. If it is not, its invocation $\mathsf{x\_sa\_propose}_i()$ terminates. Let us notice that in that case, at least x simulators have invoked $\mathsf{x\_sa\_propose}()$ on that x_safe_agreement object, and x of them are its (maybe faulty) owners.

```
operation x_sa_propose_i(v):
(01) owner_i ← X_T&S.x_compete_i();
(02) if (owner_i) then
(03)    res ← v;
(04)    for ℓ from 1 to m do
(05)        if (i ∈ SET_LIST[ℓ]) then res ← XCONS[ℓ].x_cons_propose(res) end if
(06)    end for;
(07)    X_SAFE_AG ← res
(08) end if.

operation x_sa_decide_i():
(09) wait (X_SAFE_AG ≠ ⊥);
(10) return(X_SAFE_AG).
```

Figure 6: An implementation of the x_safe_agreement type (code for simulator q_i)

If q_i is an owner, it has to cooperate with the other owners of that x_safe_agreement object in order one of the values they propose becomes the value decided from that object. To that end, these simulators could use the consensus number x underlying object that can be accessed by these x owners only. The problem is that a simulator does not know which are the other owners, and consequently does not know which is this underlying consensus number x object.

To solve this problem, two arrays are associated with each x_safe_agreement object. Let m be the number of subsets of size x in a set of n elements. We have:

- $SET\_LIST[1..m]$ is an array containing the m subsets of simulators of size x. $SET\_LIST[\ell]$ contains the subset identified by ℓ.
- $XCONS[1..m]$ is an array of m consensus number x objects. $XCONS[\ell]$ is the consensus number x object that can be accessed by the simulators that define the size x subset $SET\_LIST[\ell]$.

If simulator q_i is an owner, it scans the list $SET\_LIST[1..m]$ (all the owners have to scan it in the very same order). When it encounters a set $SETLIST[\ell]$ that contains its identity i, q_i invokes $\mathsf{x\_cons\_propose}(res)$ on the corresponding object $XCONS[\ell]$, and adopts the value it obtains from it as its current estimate of the value decided from the x_safe_agreement object. Let us notice that, whatever the set S of owners of that object, due to the systematic scanning, q_i necessarily meets S.

When it has exhausted the list, q_i deposits the value of res in X_SAFE_AG, which is then the value decided by this x_safe_agreement object.

4.4 Simulating $ASM(n,t,1)$ in $ASM(n,t',x)$: correctness proof

The proof consists in two theorems: (1) the proof that the algorithm given in Figure 6 implements the x_safe_agreement object type, and (2) the proof of the simulation itself.

Theorem 2 *The algorithm described in Figure 6 implements the* x_safe_agreement *object type.*

Proof The validity property (a decided value is a proposed value) follows directly from the text of the algorithm and the validity property of the underlying consensus number x objects $XCONS[1..m]$.

$XSAG$ being an x_safe_agreement object, let $owners[XSAG]$ be the set of its owner simulators. It is easy to see that $|owners[XSAG]| \leq x$ (see Figure 5).

For the agreement property we have to show that the invocations $XSAG$.x_sa_decide() returns the same value. For it, we show that all the writes of the underlying atomic register X_SAFE_AG (used to implement $XSAG$) do write the very same value v (line 07).

It follows from the definition of $SET\_LIST[1..m]$ that $\exists \ell$ such that $owners[XSAG] \subseteq SET\_LIST[\ell]$. Moreover, due to the agreement property of the underlying consensus number x object $XCONS[\ell]$, all the simulators that return from $XCONS[\ell]$.x_cons_propose() obtain the same value v. From then on, for any $\ell' > \ell$, any simulator in $owners[XSAG]$ proposes v to any object $XCONS[\ell']$ and (as only the simulators in $owners[XSAG]$ access these objects), it follows that only v can be decided from these objects. Hence, any simulator in $owners[XSAG]$ that executes line 07 writes v into the atomic register X_SAFE_AG, which concludes the proof of the agreement property of the x_safe_agreement object $XSAG$.

To prove the termination property of $XSAG$, let us assume that at most $(x-1)$ simulators in $owners[XSAG]$ crash while executing $XSAG$.x_sa_propose(). We have to show that any correct simulator q_i returns from its $XSAG$.x_sa_decide() invocation. As at least one simulator (q_i) in $owners[XSAG]$ is correct, it invokes $XSAG$.x_sa_propose(). As the underlying objects $XCONS[\ell]$ accessed by q_i are wait-free, q_i eventually writes a non-$\perp$ value in X_SAFE_AG. It follows that no correct simulator can block forever when it executes $XSAG$.x_sa_decide(), which concludes the proof of the theorem. $\square_{Theorem\ 2}$

Proof of the simulation As before, in order to always terminate, a t-resilient algorithm requires that at most t (simulated) processes crash. We show here that, when at most $t' \leq t \times x + (x-1)$ simulators using objects of consensus number x crash (i.e. $t \geq \lfloor \frac{t'}{x} \rfloor$), any correct simulator can simulate the code of at least $(n-t)$ simulated processes without being blocked forever in any of these codes.

In order for a simulated process p_j to decide the same value at all simulators, the snapshot values returned to p_j must be the same at all simulators, and must be consistent with the write operations.

Lemma 7 *Let $t \geq \lfloor \frac{t'}{x} \rfloor$. The simulation of at most t simulated processes can be definitely blocked (i.e., crashed) during the simulation.*

Proof The only point where the simulation of a simulated process p_j can be permanently blocked is when p_j invokes mem.snapshot(), i.e., when simulators invoke the corresponding sim_snapshot() simulation operation. As that operation involves an x_safe_agreement (invocation of x_sa_propose()), it follows that x simulators have to crash while executing the corresponding x_sa_propose() operation in order the x_safe_agreement object crash and block permanently the simulated process p_j.

On another side, due to (a) the mutex mechanism used in sim_snapshot() and (b) the dynamic determination of the x owners of a x_safe_agreement object, it follows that the crash of a simulator can participate in the crash of at most one simulated process p_j.

Hence, (a) it is necessary that x simulators crash in order a simulated process be blocked forever, and (b) these x crashes cannot block different simulated processes.

As up to t' simulators can crash, it follows that at most $\lfloor \frac{t'}{x} \rfloor \leq t$ simulated process remain definitely blocked (i.e., crash), which completes the proof of the Lemma. $\square_{Lemma\ 7}$

Lemma 8 *Let $t \geq \lfloor \frac{t'}{x} \rfloor$. Each correct simulator q_i computes the decision value of at least $(n-t)$ simulated processes p_j.*

Proof It follows from Lemma 7 that each correct simulator q_i simulates entirely the code of at least $n - \lfloor \frac{t'}{x} \rfloor \geq n-t$ simulated processes p_j. As the simulated algorithm is t-resilient, the simulation of these $n - \lfloor \frac{t'}{x} \rfloor$ simulated processes provides each of them with a decided value. Thus, each correct simulator q_i obtains decision values for at least $(n-t)$ simulated processes. $\square_{Lemma\ 8}$

Lemma 9 *The simulators that return from the simulation of the k-th invocation of mem.snapshot() issued by a simulated process p_j, obtain the same value for that simulated snapshot invocation.*

Proof The proof is the same as the proof of Lemma 3 where the type safe_agreement is replaced by x_safe_agreement. (Let us notice that both proofs rely only on the agreement property of these objects). $\square_{Lemma\ 9}$

Lemma 10 *For every simulated process p_j, the simulators decide at most one value.*

Proof The proof is similar to the one of Lemma 5. Let us observe that the mem.snapshot() operations are the only non-deterministic operations that a simulated process p_j issues. It follows from Lemma 9 that all simulators q_i obtain the same value when they invoke $\text{sim\_snapshot}_{i,j}()$, which proves the lemma. $\square_{Lemma\ 10}$

Lemma 11 *The sequences of $\text{sim\_write}_i()$, $\text{sim\_snapshot}_{i,j}()$ and $\text{sim\_x\_cons\_propose}^a_{i,j}()$ issued by each simulator (in $ASM(n,t,1)$) on behalf of each simulated processes p_j define a correct execution of the simulated algorithm.*

Proof The proof is the same as the proof of Lemma 6, where the reference to Lemma 3 is replaced by a reference to Lemma 9, and where are considered only the simulation operations sim_write() and sim_snapshot(). $\square_{Lemma\ 11}$

Theorem 3 *Let $\lfloor \frac{t'}{x} \rfloor \leq t$. The algorithms described in Figure 2, Figure 3 (where* sa_propose() *and* sa_decide() *are replaced by* x_sa_propose() *and* x_sa_decide()*), and Figure 6 are a correct simulation of $ASM(n,t,1)$ into $ASM(n,t',x)$ (for any algorithm A solving a colorless decision task in $ASM(n,t,1)$).*

Proof The proof follows from Theorem 2, Lemma 8, Lemma 10, and Lemma 11. $\square_{Theorem\ 3}$

5 Generalization

5.1 Why the simulations are for colorless tasks

A decision task can be either *colorless* or *colored*. A colored task requires that no two processes decide the same value. In a system where up to t processes may crash, it can only be guaranteed that $(n-t)$ (simulated) processes decide. Thus, if we have $(n-t')$ correct simulators with $t > t'$, we obtain less decided values than simulators. Consequently, it is not possible to provide the correct simulators with different values. This is why the previous simulation algorithms are for colorless tasks only.

5.2 $ASM(n,t',x) \simeq ASM(t+1,t,1)$

The previous simulations show that, when $t = \lfloor \frac{t'}{x} \rfloor$ (i.e., $t \times x \leq t' \leq t \times x + (x-1)$), any algorithm that solves a colorless decision task in $ASM(n,t,1)$, can be used to solve it in $ASM(n,t',x)$ and vice-versa. On another side, the BG simulation shows that any algorithm that solves a colorless decision task in $ASM(n,t,1)$ can be used to solve it in $ASM(t+1,t,1)$ and vice-versa.

It follows from the previous observations that, when $t = \lfloor \frac{t'}{x} \rfloor$, any algorithm that solves a colorless decision task in $ASM(n,t',x)$ can be used to solve it in $ASM(t+1,t,1)$, and vice-versa. This equivalence is denoted $ASM(n,t',x) \simeq ASM(t+1,t,1)$.

5.3 General case: $ASM(n_1,t_1,x_1) \simeq ASM(n_2,t_2,x_2)$

Assuming $n_1 > t_1$ and $n_2 > t_2$, due to the simulation described in Section 3 and Section 4 we have $ASM(n_1,t_1,x_1) \simeq ASM(n_1, \lfloor \frac{t_1}{x_1} \rfloor, 1)$ and $ASM(n_2,t_2,x_2) \simeq ASM(n_2, \lfloor \frac{t_2}{x_2} \rfloor, 1)$.

Let us assume that $\lfloor \frac{t_1}{x_1} \rfloor = \lfloor \frac{t_2}{x_2} \rfloor = t$. Due to the BG simulation we have then $ASM(n_1,t,1) \simeq ASM(t+1,t,1)$ and $ASM(n_2,t,1) \simeq ASM(t+1,t,1)$. If follows that, by transitivity, we have $ASM(n_1,t_1,x_1) \simeq ASM(n_2,t_2,x_2)$. This equivalence is depicted in Figure 7.

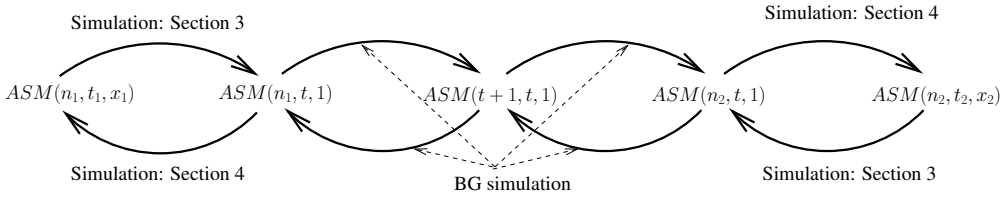

Figure 7: Model equivalence (for $\lfloor \frac{t_1}{x_1} \rfloor = \lfloor \frac{t_2}{x_2} \rfloor$)

5.4 Increasing the consensus number can be useless

Playing with the model parameters Let us consider the models $ASM(n,t,x)$ and $ASM(n,t,x+\Delta x)$. It follows that, albeit the second one has stronger objects than the first one, it is not more powerful if $\lfloor \frac{t}{x} \rfloor = \lfloor \frac{t}{x+\Delta x} \rfloor$. Hence, considering base objects with higher consensus number does not always increase the power of the model. Similarly, increasing the upper bound on the number of faulty processes from t to $t+\Delta t$ does not make the system model weaker if $\lfloor \frac{t}{x} \rfloor = \lfloor \frac{t+\Delta t}{x} \rfloor$.

This exhibits the *multiplicative power of consensus numbers*, which states that $ASM(n,t,1) \simeq ASM(n,t',x)$ for $t \times x \leq t' \leq (t \times x) + (x-1)$.

Equivalence classes More generally, this shows that the whole set of system models can be partitioned into equivalence classes. All the models $ASM(n, t', x)$ such that $\lfloor \frac{t'}{x} \rfloor = t$ belong to the same class (from a computability power point of view) and $ASM(n, t, 1)$ can be taken as the canonical form representing all the models of that class.

As an example let us fix $t' = 8$. We have:

- All the system models $ASM(n, 8, x)$, for $9 \leq x \leq n$, have the same power as $ASM(n, 0, 1)$.
- The 4 system models $ASM(n, 8, x)$, for $5 \leq x \leq 8$, have the same power as $ASM(n, 1, 1)$.
- Both the system models $ASM(n, 8, 4)$, $ASM(n, 8, 3)$ have the same power as $ASM(n, 2, 1)$.
- The system model $ASM(n, 8, 2)$ has the same power as $ASM(n, 4, 1)$.
- The last class is $ASM(n, 8, 1)$.

More generally, we have the following: if $\frac{t'}{t} \geq x > \frac{t'}{t+1}$ then $ASM(n, t', x) \simeq ASM(n, t, 1)$. This means that, given t and t', all the objects whose consensus numbers x_1, x_2, etc., satisfy the previous inequalities are equivalent in a system of n processes prone to t' crashes.

A hierarchy of system models As indicated in the introduction, Gafni and Kuznetsov have introduced the notion of *set consensus number* associated with a task [17]. The set consensus number of a task T is k if T cannot be wait-free solved in an asynchronous system where processes have access to $(k+1)$-set agreement objects, but can be wait-free solved when the processes have access to k-set agreement objects. In a system of n processes, the set consensus numbers define a size n hierarchy between tasks. Class 1 consists of universal tasks (because they can solve consensus and consequently any other task), while class n contains the trivial tasks (the tasks that can be solved asynchronously) [17]. More generally, a task in class k is more difficult than a task in class $k+1$.

The k-set agreement problem is impossible to solve in $ASM(n, k, 1)$ [6, 22, 28], but can be solved in $ASM(n, k-1, 1)$ [11]. It follows that any task with consensus number k can be solved in $ASM(n, k-1, 1)$ and cannot be solved in $ASM(n, k, 1)$. This establishes the following hierarchy among systems: a system model $\mathcal{S}$ is stronger than a system model $\mathcal{S}'$ ($\mathcal{S} \succ \mathcal{S}'$), if more tasks can be solved in $\mathcal{S}$ than in $\mathcal{S}'$ (e.g., $ASM(n, 3, 1) \succ ASM(n, 4, 1)$: 4-set agreement can be solved in $ASM(n, 3, 1)$ but not in $ASM(n, 4, 1)$).

The relation between a task T_k, with set consensus number k, and a system model $ASM(n, t, x)$ is then the following: T_k can be solved in $ASM(n, t, x)$ if and only if $k > \lfloor \frac{t}{x} \rfloor$. Because the set consensus numbers define a hierarchy, the preceding relation establishes a hierarchy among system models.

6 Conclusion

This paper was on the computability power of systems made up of n processes that communicate by accessing read/write shared registers and objects with consensus number x with $x \leq t$, where t is the maximal number of processes that may crash in a run. It has shown that, as far as colorless decision tasks are concerned, we have the following equivalences:

- $ASM(n, t', x) \simeq ASM(n, t, 1)$ iff $\lfloor \frac{t'}{x} \rfloor = t$, i.e., $(t \times x) \leq t' \leq (t \times x) + (x - 1)$.
- $ASM(n, t, x) \simeq ASM(t + 1, t, x)$.
- $ASM(n_1, t_1, x_1) \simeq ASM(n_2, t_2, x_2)$ iff $\lfloor \frac{t_1}{x_1} \rfloor = \lfloor \frac{t_2}{x_2} \rfloor$.

Hence, when we consider the computability power of an asynchronous shared memory system prone to process crashes, the BG simulation is one side of the coin, while the multiplicative power of consensus numbers is the other side of the coin.

The BG simulation has recently been extended from colorless tasks to colored tasks (such as the renaming problem where no two processes are permitted to decide the same new name [3]) [16, 23]. Hence the following noteworthy question: Is it possible to extend the proposed simulation algorithms to colored decision tasks? This problem, that remains open, constitutes a nice research challenge for the community.

Acknowledgments

The authors want to thank E. Gafni, S. Rajsbaum, and C. Travers for interesting discussions on the BG-simulation.

References

[1] Afek Y., Attiya H., Dolev D., Gafni E., Merritt M. and Shavit N., Atomic Snapshots of Shared Memory. *Journal of the ACM*, 40(4):873-890, 1993.

[2] Afek Y., Gafni E., Rajsbaum S., Raynal M. and Travers C., The k-Simultaneous Consensus Problem. *Distributed Computing*, To appear, 2010, DOI 10.1007/s00446-009-0090-8.

[3] Attiya H., Bar-Noy A., Dolev D., Peleg D. and Reischuk R., Renaming in an Asynchronous Environment. *Journal of the ACM*, 37(3):524-548, 1990.

[4] Attiya H. and Rachman O., Atomic Snapshots in O($n \log n$) Operations. *SIAM Journal on Computing,* 27(2):319-340, 1998.

[5] Attiya H. and Welch J., Distributed Computing, Fundamentals, Simulation and Advanced Topics (Second edition). *Wiley Series on Parallel and Distributed Computing*, 414 pages, 2004.

[6] Borowsky E. and Gafni E., Generalized FLP Impossibility Results for t-Resilient Asynchronous Computations. *Proc. 25th ACM Symposium on Theory of Computing (STOC'93)*, ACM Press, pp. 91-100, 1993.

[7] Borowsky E. and Gafni E., The Implication of the Borowsky-Gafni Simulation on the Set Consensus Hierarchy. *Tech Report CSD-930021*, 7 pages, UCLA, 1993.

[8] Borowsky E., Gafni E., Lynch N. and Rajsbaum S., The BG Distributed Simulation Algorithm. *Distributed Computing*, 14(3):127-146, 2001.

[9] Castañeda A. and Rajsbaum S., New Combinatorial Topology Upper and Lower Bounds for Renaming. *Proc. 27th ACM Symposium on Principles of Distributed Computing (PODC'08)*, ACM Press, pp. 295-304, Toronto (Canada), 2008.

[10] Chandra T., Hadzilacos V. and Toueg S., The Weakest Failure Detector for Solving Consensus. *Journal of the ACM*, 43(4):685-722, 1996.

[11] Chaudhuri S., More *Choices* Allow More *Faults:* Set Consensus Problems in Totally Asynchronous Systems. *Information and Computation,* 105:132-158, 1993.

[12] Chaudhuri S. and Reiners P., Understanding the Set Consensus Partial Order Using the Borowsky-Gafni Simulation. *Proc. 10th Int'l Workshop on Distributed Algorithms (WDAG'96, now DISC)*, Springer Verlag LNCS #1151, pp. 362-379, 1996.

[13] Delporte-Gallet C., Fauconnier H., Guerraoui R., Hadzilacos V., Kouznetsov P. and Toueg S., The Weakest Failure Detectors to Solve Certain Fundamental Problems in Distributed Computing. *Proc. 23th ACM Symposium on Principles of Distributed Computing (PODC'04)*, ACM Press, pp. 338-346, 2004.

[14] Fischer M.J., Lynch N.A. and Paterson M.S., Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374-382, 1985.

[15] Gafni E., Round-by-round Fault Detectors: Unifying Synchrony and Asynchrony. *Proc. 17th ACM Symposium on Principles of Distributed Computing (PODC'98)*, ACM Press, pp. 143-152, 1998.

[16] Gafni E., The Extended BG Simulation and the Characterization of t-Resiliency. *Proc. 41th ACM Symposium on Theory of Computing (STOC'09)*, ACM Press, pp. 85-92, 2009.

[17] Gafni E. and Kuznetsov P., On Set Consensus Numbers. *Proc. 23th Int'l Symposium on on Distributed Computing (DISC'09)*, Springer Verlag LNCS #5805, pp. 35-47, 2009.

[18] Gafni E., Raynal M. and Travers C., Test&set, Adaptive Renaming and Set Agreement: a Guided Visit to Asynchronous Computability. *26th IEEE Symposium on Reliable Distributed Systems (SRDS'07)*, IEEE Computer Press, pp. 93-102, 2007.

[19] Guerraoui R. and Kuznetsov P., Failure Detectors as Type Boosters. *Distributed Computing*, 20:343-358, 2008.

[20] Herlihy M.P., Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124-149, 1991.

[21] Herlihy M.P. and Rajsbaum S., Algebraic Spans, *Mathematical Structures in Computer Science*, 10(4): 549-573, 2000.

[22] Herlihy M., Shavit N., The Topological Structure of Asynchronous Computability. *Journal of the ACM*, 46(6):858-923, 1999.

[23] Imbs D. and Raynal M., Visiting Gafni's Reduction Land: From the BG Simulation to the Extended BG Simulation. *Proc. 11th Int'l Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'09)*, Springer Verlag LNCS #5873, pp. 369-383, 2009.

[24] Loui M.C., and Abu-Amara H.H., Memory Requirements for Agreement Among Unreliable Asynchronous Processes. *Par. and Distributed Computing: vol. 4 of Advances in Comp. Research,* JAI Press, 4:163-183, 1987.

[25] Lynch N.A., Distributed Algorithms. *Morgan Kaufmann Pub.*, San Francisco (CA), 872 pages, 1996.

[26] Mostéfaoui A., Raynal M. and Travers C., Narrowing Power vs Efficiency in Synchronous Set Agreement: Relationship, Algorithms and Lower Bound. *Theoretical Computer Science*, 411:58-69, 2010.

[27] Neiger, G., Failure Detectors and the Wait-free Hierarchy. *Proc. 14th ACM Symposium on Principles of Distributed Computing (PODC'95)*, ACM Press, pp. 100-109, 1995.

[28] Saks M. and Zaharoglou F., Wait-Free k-Set Agreement is Impossible: The Topology of Public Knowledge. *SIAM Journal on Computing*, 29(5):1449-1483, 2000.