Advance Access published on August 3, 2006 doi:10.1093/comjnl/bxl046

The Alpha of Indulgent Consensus

RACHID GUERRAOUI[1] AND MICHEL RAYNAL[2,*]

[1]*Distributed Programming Lab., EPFL, Lausanne, Switzerland*
[2]*IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France*
[*]*Corresponding author: michel.raynal@irisa.fr*

This paper presents a simple framework unifying a family of consensus algorithms that can tolerate process crash failures and asynchronous periods of the network, also called indulgent consensus algorithms. Key to the framework is a new abstraction we introduce here, called *Alpha*, and which precisely captures consensus safety. Implementations of *Alpha* in shared memory, storage area network, message passing and active disk systems are presented, leading to directly derived consensus algorithms suited to these communication media. The paper also considers the case where the number of processes is unknown and can be arbitrarily large.

Keywords: asynchronous system, disk, consensus, message-passing, shared memory

Received 28 July 2005; revised 3 July 2006

1. INTRODUCTION

Distributed computing is about devising algorithms for a set of logical entities representing individual Turing machines, and usually called processes. It is common in distributed computing to assume that the processes communicate through a communication medium that does not corrupt shared information. (The issue of information corruption is typically tackled separately through cryptographic techniques.) The communication medium can be made up of a shared memory, a storage area network, message passing channels or active disks. Not surprisingly, the ability of the processes to reach consensus on a common decision based on possibly different proposals is key to distributed computing. If no agreement is ever needed in the computation, then this consists of a set of centralized independent programs rather than of a global distributed program.

Informally, the consensus problem can be described as follows. Each process proposes a value, and each correct process (one that does not crash) has to decide a value (termination), in such a way that there is a single decided value (agreement) and that value is a proposed value (validity). A seminal result in the theory of distributed computing is the impossibility of solving consensus in an asynchronous system even if only one process may crash [1]. The impossibility stems from the very fact that an asynchronous system has no timing bounds on process speed or communication delays. In particular, in an asynchronous system, any algorithm that ensures validity and agreement would have at least one execution where it does not terminate.

In practice, however, distributed systems are usually synchronous and do have timing bounds on process speeds and communication delays. Sometimes, due to high contention, a distributed system might traverse an 'instability' period during which the bounds are violated, before resuming to a 'stable' period during which some bounds do hold.

Synchronous consensus algorithms are pretty easy to design. These are however fragile and might violate the safety property of consensus (during 'instability' periods of the system). Appealing alternatives are consensus algorithms that preserve the safety of consensus during 'instability' periods, and achieves liveness whenever the system 'stabilizes'. Such algorithms have been said to be *indulgent* [2]. These algorithms are usually devised in partially synchronous models [3] with only eventual timing assumptions.

There has been a lot of work to precisely capture these eventual timing assumptions in the form of an abstract *liveness-oriented* device. Most approaches came out from a closer look at the consensus impossibility, revealing that the need for timing assumptions was basically motivated by the necessity to detect failures and distinguish crashed processes from slow ones. Not surprisingly, the minimal timing assumptions needed to solve consensus were captured by an abstract device sometimes called a *failure detector* [4, 5, 6]. From a theoretical point of view, this device is defined through axiomatic properties and the system enriched with these properties is defined as a new computation model. The most popular of these devices, called *Omega* (or sometimes eventual leader election), was indeed shown to be minimal to solve consensus [5]. From a practical perspective, exhibiting a device like *Omega* makes it easier to design different forms of indulgent consensus algorithms by varying the implementations of *Omega*, as long as its properties remain satisfied.

By itself, *Omega* does not implement consensus. It is rather complementary to an algorithm that preserves consensus safety even during instability periods. A variety of such algorithms have been devised and more will certainly be discovered in the near future. This paper was motivated by the appealing objective to devise the complement to *Omega*. That is, to capture the exact properties of the algorithm that, together with *Omega*, implement consensus (in an indulgent way). The result of this quest was precisely *Alpha*, namely, the safety-oriented abstraction we present in this paper.

Contribution This paper presents a simple and generic framework that instantiate indulgent consensus algorithms that clearly decouple safety and liveness properties. Liveness relies on the use of the *Omega* abstraction. The information structure used to ensure safety is encapsulated within the *Alpha* abstraction we introduce here.

Interestingly, our generic framework is independent from the way the processes communicate. Communication is encapsulated within the implementation of *Alpha*. Several implementations of *Alpha* are presented, each considering a specific communication medium. In particular, the paper visits the following communication models: shared memory, storage area network, message passing channels and active disks. We also consider the case where the number of processes is not known and can be arbitrary large [7, 8, 9].

Our paper is driven by *design simplicity* and an interesting methodological feature of the paper is the way these algorithms are '*derived*' one from the other.[1] We start the visit from a shared memory model providing basic register objects [10, 11]. Then, using classic quorum techniques, we show how each shared register can be replaced with shared disk blocks [11]. Then, considering n disks (the same as the number of processes), and observing that each process p_i can act as a peer playing two roles, a role offering the *Alpha* primitive to its user, and a disk storage role, we obtain a message-passing algorithm [12]. Finally, restricting a process to play a single role (either implementing the *Alpha* primitive or being only a disk storage), we obtain an active disk-based algorithm [8].

Related work The spirit of our *Alpha* abstraction (implicitly hidden in [12] and inspired from the notion of 'eventual register' introduced in [13]) is close to the one of the *Lambda* abstraction we have investigated in [14]. Both are designed for round-based algorithms, and defined in an abstract way by a set of axiomatic properties. However, unlike *Lambda*, *Alpha* does not encapsulate failure detection issues that are needed to terminate consensus; it captures the essence of what is exactly needed as far as consensus safety is concerned [12]. It is in this sense a strict complementary to *Omega* (as far as consensus is concerned). We thus achieve a clean separation of liveness and safety, in the spirit of indulgent algorithms.

Alpha differs from the notion of eventual register [13, 15] (dedicated to the deconstruction or reconstruction of Paxos algorithms), or the subsequent notion of ranked register [8] (introduced to take into account active disks), in the sense that it is a higher level abstraction providing a simple unifying abstraction factoring out the way the processes communicate and cooperate (whatever the underlying communication medium). By analogy to failure detectors that are defined in an abstract way independently of any particular implementation detail (such as message delay, local clocks, network topology etc.), *Alpha* provides a unique framework to describe a wide variety of indulgent consensus algorithms.

The algorithms we obtain by instantiating our framework can be viewed as variants of the seminal Lamport's Paxos algorithm [12], and more precisely of its underlying Synod consensus algorithm. (We focus in this paper on variants in a crash-stop model with reliable communication, for simplicity of presentation. Extensions to the crash-recovery and failure omission model can also be obtained through our framework). The goal of Paxos is actually to implement highly available deterministic services despite faulty processes. Paxos is based on the well-known *state machine* approach [16] (also called *active replication*): the service is replicated over a set of processes, and every replica is supposed to compute every request and return the associated result to the corresponding client (which selects the first returned result). It is crucial to Paxos that the replicas deliver client's requests in the same total order. The replicas can thus apply the same sequence of requests to their local copies of the service state. In that way, the processes create the illusion that there is a single copy of the service. Paxos ensures the common request delivery order by using a sequence of consensus instances, each instance ordering a batch of requests.[2] Paxos has been analyzed and described in various ways in several papers [13, 15, 17, 18, 19]. Moreover, several protocols inspired by or based on Paxos have been recently developed [20, 8, 11]. Indirectly, our paper can be seen as an endeavor to capture the essence of Paxos-like consensus algorithms (e.g. [13, 8, 18]).

Roadmap The paper is made up of nine sections. Section 2 presents the process model. Section 3 presents the *Alpha* abstraction and a generic consensus framework. Then the Sections 4, 5, 6 and 7 present implementations of *Alpha* suited to different underlying communication systems, namely shared memory, disks, message-passing and active disks. Finally, Section 8 addresses the case where there are arbitrarily many processes. Section 9 concludes the paper.

[1] In the present context, the very notion of 'derived' does not mean that the algorithms are obtained from automatic transformations. It means that they are obtained from a methodological construction.

[2] Although the original name of the consensus protocol underlying Paxos was the 'Synod' algorithm, this consensus algorithm was also named 'Paxos' in the literature.

2. PRELIMINARIES

2.1. Processes

We consider a finite set of $n > 1$ processes $p_1, p_2, \ldots, p_n$. A process can fail by *crashing*, i.e. prematurely halting. Until it possibly crashes, a process behaves according to its specification and executes atomic computation steps. A step consists in reading or writing a local variable, invoking an operation on a shared service (e.g. *Omega*), or an execution of the `return`() statement. If it crashes, a process stops executing any step. A run is a sequence of steps issued by the processes.

By definition, a process is *faulty* during a run if it crashes during that run. Otherwise, it is *correct* in that run. In the following, t denotes the maximum number of processes that may crash.

There is no assumption on the relative speed of a process with respect to another. We only assume that, until it possibly crashes, the speed of a process is positive (it cannot stop during an infinite period between two consecutive steps of its algorithm).

We assume that the communication medium through which the processes communicate is reliable. Reliability means here that the medium does not corrupt data. The relevant properties of a particular communication medium will be described when we will present a specific instance of our framework for that particular medium.

2.2. The consensus problem

In the *consensus* problem, every process p_i is supposed to *propose* a value v_i and the processes have to *decide* on the same value v, which has to be one of the proposed values. More precisely, the problem is defined by two safety properties (*validity* and *uniform agreement*) and a liveness property (*termination*):

- `Validity`: If a process decides v, then v was proposed by some process.
- `Agreement`: No two processes decide differently.
- `Termination`: Every correct process eventually decides on some value.

In the following $\perp$ denotes a default value that cannot be proposed by a process.

2.3. The Omega abstraction

The *Omega* abstraction, denoted Ω, and sometimes called *eventual leader election*, provides the processes with an operation `Omega`() that returns the value *true* or *false* each time it is invoked by a process. When the invoking process p_i obtains the value *true*, we say that it is currently elected. A unique correct leader is eventually elected but there is no knowledge of when the unique correct leader is elected. Several leaders can coexist during an arbitrarily long period of time, and there is no way for the processes to learn when this 'anarchy' period is over. The *Omega* abstraction satisfies the following property (that property refers to a notion of global time, but this notion is not accessible to the processes):

- `Eventual Leadership`: There is a time τ and a correct process p_i such that, after τ, every invocation of `Omega`() by p_i returns *true*, and any invocation of `Omega` () by $p_j \neq p_i$ returns *false*.

The *Omega* abstraction has been introduced and formally developed in [5]. According to its original specification in [5], *Omega* is invoked at every computation step of a process and returns the identity of a process which is said to be *trusted* : eventually the same correct process is trusted by all correct processes. For presentation simplicity, but without loss of generality, we considered a slightly different specification here where (a) we do not force a process p to invoke that abstraction at every step of its computation and (b) the abstraction simply returns a boolean according to whether the process p is trusted or not. *Omega* was shown to be the weakest, in terms of information about failures, to solve consensus in a distributed system prone to process crashes but where a majority of processes are correct (which is the best, lower bound, that can be attained with *Omega* [5]). Several *Omega*-based consensus protocols have been proposed, e.g., [22, 14, 12, 23] for message-passing systems, and [24] for shared memory systems. All these protocols are indulgent [2]. It is important to notice that the original version of Paxos was not described using the *Omega* abstraction; it was rewritten using the *Omega* abstraction in [13].

Omega cannot be implemented in purely asynchronous systems: this would violate the impossibility of solving consensus despite asynchrony and failures [1]. To implement *Omega*, one has to make additional assumptions. Such assumptions can take the form of eventual synchrony properties. Several protocols implementing *Omega* in message passing systems enriched with such additional properties have been proposed (e.g., [25] and [26] consider the system has eventual timely channels, while [27] assumes that the message exchange pattern eventually satisfies some ordering property). A protocol implementing *Omega* that combines timing assumptions and time-free assumptions is described in [28]. An *Omega* protocol for a shared memory system (the only we are aware of) can be found in [21]. (In contrast, as we will see in the rest of the paper, the other basic abstraction, *Alpha*, can be implemented in fully asynchronous systems.)

3. A GENERIC FRAMEWORK

3.1. The alpha abstraction

Alpha captures the essence of indulgent consensus as far as consensus safety is concerned. In short, one can view the

Alpha abstraction as a shared one-shot storage object: processes invoke it with a value to store and get a value in return: the actual value stored. If accessed concurrently, the object might not store anything (it sticks with its initial value $\perp$). If accessed sequentially, the object stores the first value and holds it forever.

More specifically, the *Alpha abstraction* (we also say *Alpha object* when we consider one of its instances) exports a single operation, denoted `Alpha()`. Each time a process p_i invokes `Alpha()`, p_i provides it with a pair of parameters, namely, a round number r and a value v. *Alpha* assumes that (1) distinct processes use distinct round numbers, and (2) each process uses strictly increasing round numbers.[3] In the following, when we write `Alpha` $(r, -)$, we mean that the second parameter of the invocation is irrelevant for the property we consider. The *Alpha* abstraction is defined by the following set of properties.

- `Validity`.
  If the invocation `Alpha` (r, v) returns, the returned value is either $\perp$ or a value v' such that there is a round $r' \leq r$ and `Alpha` (r', v') has been invoked by some process.
- `Quasi-agreement`.[4]
  Let `Alpha` $(r, -)$ and `Alpha` $(r', -)$ be any two invocations that return v and v' respectively. We have, $((v \neq \perp) \wedge (v' \neq \perp)) \Rightarrow (v = v')$.
- `Conditional non-`$\perp$` convergence`.
  An invocation $I =$ `Alpha` $(r, -)$ must return a non-$\perp$ value if every invocation $I' =$ `Alpha` $(r', -)$ that starts before I returns is such that $r' < r$.
- `Termination`.
  Any invocation `Alpha()` by a correct process returns.

As we will see more explicitly when we will use it, it is important to notice *Alpha* is a safety-oriented abstraction: its aim is to guarantee the validity and agreement properties of consensus (i.e. its safety properties). It is also important to notice that the *Alpha* abstraction (considered alone) is not powerful enough to ensure the consensus termination property (this is because concurrent processes might never be able to store any value in an *Alpha* object). So, the termination property of the `Alpha()` operation is not related to consensus liveness; it only states that, similarly to a read or write operation, an `Alpha()` invocation has to terminate.

3.2. A generic algorithm

Algorithm description A generic consensus algorithm based on the abstractions *Alpha* and *Omega* is described in Figure 1. For simplicity of exposition, this description uses a shared

```
function consensus (v):
(1) r ← 0;
(2) while (DECIDED = ⊥) do
(3)     if [Omega()] then res ← [Alpha (r + i, v)] ;
(4)                  if res ≠ ⊥ then DECIDED ← res
(5)                             else r ← r + n
(6)                  end_if
(7)     end_if
(8) end_while;
(9) return (DECIDED)
```

FIGURE 1. A generic framework, code for process p_i $(t < n)$.

atomic variable *DECIDED* (initialized to $\perp$) whose aim is to contain the decided value.[5] The simplicity and elegance provided by the use of *Alpha* and *Omega* is conveyed by the figure. As we pointed it out, the safety issue addressed by the *Alpha* abstraction is clearly separated from the liveness issue solved by the leader abstraction *Omega*.

The algorithm is round-based and the intuitive idea is the following. Processes go from a round to a higher round and access *Alpha* in every round they move to. To ensure that *Alpha* indeed stores a value that will be the consensus decision value and prevent contention among processes, processes use rounds in a disciplined fashion. Basically, a round is kind of 'resource' that has to be used eventually by a single process (i.e. in a very unfair way!). The leader abstraction *Omega* plays precisely the role of resource allocator providing the required 'eventual unfairness'. A process p_i does move to the next round only with the permission of *Omega* (i.e. if p_i is leader). Eventually, a single process keeps accessing *Alpha* and does so with a round that is higher for a non-$\perp$ value to be returned.

A process p_i invokes *consensus* (v_i) where v_i is the value it proposes to the consensus instance. It terminates its participation to that instance when it executes return (*DECIDED*) at line 9; moreover, p_i uses the sequence of increasing round numbers i, $i + n$, $i + 2n$ etc. (so, no two processes use the same round numbers). The local variable r is used by p_i to keep track of round numbers. If p_i considers it is leader, it invokes `Alpha` $(r + i, v_i)$. According to the result value it obtains, p_i helps the other processes decide (by storing in *DECIDED* the non-$\perp$ value it is about to decide), or starts a new loop (waiting for a decided value or to be again elected leader). The text of the algorithm is self-explanatory.

If a single correct leader is elected from the very beginning, consensus is obtained after the first invocation of `Alpha()` by the leader. Moreover, in that case, the (message and time) cost of the algorithm does not depend on the number of faulty processes.

Correctness Proof Due to the tests of line 2 and line 4, a consensus function cannot return $\perp$. The validity property

[3] Round numbers are called ballots in Paxos 2.

[4] This property is called quasi-agreement instead of agreement to emphasize the fact that two values can be returned, namely, a non-$\perp$ value proposed by a process, and the default value $\perp$.

[5] By convention, we use uppercase letters for shared variables and lowercase letters for local variables.

of consensus follows from this observation, the validity property of *Alpha*, and the fact that a process always invokes `Alpha()` with the value it proposes. The consensus agreement property follows from the fact that $\perp$ cannot be decided, combined with the quasi-agreement property of the *Alpha* abstraction.

The consensus termination property follows from the eventual leadership guarantee provided by *Omega*. To show that there is a process that eventually stores a value in *DECIDED*, assume by contradiction that no process stores a value in *DECIDED*. By *Omega*, there is a time τ after which a correct process (say p_ℓ) is forever elected as single leader, which means that, from τ on, only p_ℓ can execute 'non empty' rounds (i.e. lines 3–6) and invoke `Alpha()`. As the round numbers used by a process can only increase, p_ℓ eventually executes a round r such that, any other process that invoked `Alpha()` with some round number r' is such that $r > r'$. Due to the conditional non-$\perp$ convergence and termination properties of *Alpha*, we conclude that `Alpha` (r, v_i) returns a non-$\perp$ value that is deposited in *DECIDED*: a contradiction.

Remark It is also important to remark that our proof is only based on the properties provided by the abstractions *Omega* and *Alpha*. There is no additional requirement on the number (denoted t) of processes that are allowed to crash. So it works for $t < n$. In a sense, these requirements are encapsulated within the abstractions, and in particular *Alpha*.

4. IMPLEMENTING ALPHA IN A SHARED MEMORY SYSTEM

This section presents an implementation of an *Alpha* object in a shared memory model.

4.1. Shared memory model

The shared memory is made up of an array of n reliable 1WnR (one writer/n readers) regular registers, denoted *REG*[1..n]. 1 WnR means that the register *REG*[i] can be7 read by any process and written only by p_i. Reliability means here that a register never crashes: it can always execute a read or a write operation and never corrupts its value. Regularity means the following [29]: a regular register is a shared register such that a read that is not concurrent with a write (their executions do not overlap) delivers the current value of the register; a read concurrent with one or more—sequential—writes delivers the previous value of the register or one of the values being written.

The notion of regular register has been introduced by Lamport [29]. A *regular* register is weaker than an *atomic* register in the following sense. Consider a register with initial value v. Let $R1$ and $R2$ be two consecutive read operations ($R2$ starts after $R1$ has completed) issued by the same or two different processes, and a write W that writes v' and that is concurrent with $R1$ and $R2$. If the register is regular, it is possible for the first read $R1$ to obtain the second value v', while the second read $R2$ obtains the initial value v. This is called a 'new/old inversion'. (An atomic register is a regular register that does not allow for new/old inversions [29]. If the first read operation $R1$ obtains the second value v', no subsequent read R_2 can obtain a value older than v').

A register *REG*[i] is made up of three fields *REG*[i].*lre*, *REG*[i].*lrww*, and *REG*[i].*val*, initialized to $0, -i$ and $\perp$ respectively. The meaning of these fields is the following:

- *REG*[i].*lre* stores the number of the *l*ast *r*ound *e*ntered by p_i. It can be seen as the logical date of the last invocation of `Alpha()` issued by p_i.
- *REG*[i].*lrww* and *REG*[i].*val* constitutes a pair of related values 0: *REG*[i].*lrww* stores the number of the *l*ast *r*ound *w*ith a *w*rite of a value in the field *REG*[i].*val*. So, *REG*[i].*lrww* is the logical date of the last write in *REG*[i].*val*, that contains the value that p_i attempts to write in the *Alpha* object.

To simplify the writing and the reading operations in the algorithm, we consider that each field of *REG*[i] can be written separately. This is done without loss of generality because, as the process p_i is the only one that can write *REG*[i], it trivially knows its last value. So, *REG*[i].*lre* $\leftarrow r$ is a shortcut for *REG*[i]$\leftarrow \langle r,$ *REG*[i].*lrww*, *REG*[i].*val*$\rangle$ and, similarly, *REG*[i].(*lrww*, *val*) $\leftarrow$ (r, value) stands for *REG*[i] $\leftarrow \langle r, r,$ value$\rangle$.

4.2. The algorithm

The algorithm implementing `Alpha` (r, v) using a shared memory is described in Figure 2 (it is close to algorithms that can be found in [10, 11]). A simple examination of its code shows that it is a wait-free algorithm: if p_i does not crash while executing `Alpha` (r, v) it will terminate (at line 3, 8 or 9) whatever the behavior of the other processes.

A naive algorithm To get an intuition of the algorithm, let us consider the particular case where there would be no concurrent invocations of `Alpha` (−). The implementation would then be very simple. A register could have only one field, namely, *REG*[i].*val*, and a process would only have to execute the following statements (*reg*[1..n] is a local array destined to contain the result of reading *REG*[1..n]):

reg[1..n] $\leftarrow$ *REG*[1..n]; % p_i reads (in any order) the regular registers % `if` ($\exists j$:*reg*[j].*val*$\neq\perp$) **then** *value* $\leftarrow$ *reg*[j].*val* **else** *value* $\leftarrow v$ **end_if**; *REG*[i].*val* $\leftarrow$ *value*;
`return` (*value*)

Coping with concurrency The problems are created by concurrent `Alpha()` invocations. (Of course addressing concurrency by always returning $\perp$ would violate the conditional non-$\perp$ convergence!) So, to preserve both quasi-agreement and conditional non-$\perp$ convergence despite concurrent invocations, the algorithm is based on the

```
function Alpha (r, v): % This is the text for p_i %

   % Step 1 ---------------------------------------------------------
   % 1.1: p_i first makes public the date of its last attempt %
(1) REG[i].lre ← r;
   % 1.2: Then, p_i reads the shared registers to know the other processes progress %
(2) reg[1..n] ← REG[1..n]; % p_i reads (in any order) the regular registers %
   % 1.3: p_i aborts its attempt if another process started a higher round %
(3) if (∃j : reg[j].lre > r) then return (⊥) end_if;
   % Step 2 ---------------------------------------------------------
   %Then p_i adopts the last value that has been deposited in a register %
   % If there is no such value, it adopts its own value v %
(4) let value be reg[j].val where j is such that ∀k : reg[j].lrww ≥ reg[k].lrww;
(5) if (value = ⊥) then value ← v end_if;
   % Step 3 ---------------------------------------------------------
   % 3.1. p_i writes the value it has adopted (together with the current date) %
(6) REG[i].(lrww, v) ← (r, value);
   % 3.2: p_i reads again the shared registers to know the processes's progress %
(7) reg[1..n] ← REG[1..n];
   % 3.3: And aborts its attempt if another process started a higher round %
(8) if (∃j : reg[j].lre > r) then return (⊥) end_if;
   % Step 4 ---------------------------------------------------------
   % Otherwise, value is the result of the Alpha abstraction: p_i returns it %
(9) return (value)
```

FIGURE 2. Alpha in a shared memory system.

following idea: we use logical time to timestamp the invocations, and answer $\perp$ when the corresponding invocation is not processed according to the total order defined by the timestamps.[6] The algorithm uses the round numbers as logical dates associated with each invocation (notice that no two invocations have the same dates). Intuitively, the algorithm aims at ensuring that, if there is a last invocation of `Alpha()` ('last' with respect to the round numbers), it will succeed in associating a definitive value with the *Alpha* object.[7] To that aim, the algorithm manages and uses control information, namely the 'date' fields of each shared register (i.e. *REG*[*i*].*lre* and *REG*[*i*].*lrww*).

More explicitly, a process p_i proceeds as follows:

- Step 1 (lines 1–3).
  - Line 1: When it starts executing `Alpha` (*r*, *v*), p_i first informs the other processes that the *Alpha* object has attained (at least) the date *r*
  - Line 2: Then p_i reads the shared registers to know the 'current state' of the other processes.
  - Line 3: If it discovers it is late (i.e. other processes have invoked `Alpha` (−) with higher dates), p_i aborts its current attempt and returns $\perp$. Let us observe that this preserves quasi-agreement and does not contradict conditional non-$\perp$ convergence.
- Step 2 (lines 4–5). If, it is not late, p_i determines a value. To not violate quasi-agreement, p_i selects the last value ('last' according to the round numbers/logical dates) that has been deposited in a regular register *REG*[*j*]. If there is no such value it considers its own value *v*. If there is one it is unique because no two processes use the same round numbers.
- Step 3 (lines 6–8).
  - Line 6: p_i writes in *REG*[*i*] the value it has computed (together with its timestamp).
  - Lines 7–8: It reads again the shared registers to check again if it is late (in that case, there are concurrent invocations of `Alpha()` with higher dates). As before, if it is the case, p_i aborts its current attempt and returns $\perp$.
- Step 4 (line 9).
- Otherwise, p_i was not late: it actually succeeded in 'writing' *v* in the *Alpha* object and consequently returns that value.

4.3. Correctness proof

Termination As it is wait-free, the algorithm described in Figure 2 trivially satisfies the termination property.

Validity Let us observe that if a value *v* is written in *REG*[*i*].*val*, that value has been previously passed as a parameter in an `Alpha()` invocation (lines 4–6). The validity

[6] A similar idea has been used in timestamp-based transaction systems [32]. A timestamp is associated with each transaction, and a transaction is aborted when it accesses a data that has already been accessed by another transaction with a higher timestamp (an aborted transaction has to be re-issued with a higher timestamp).

[7] Thanks to the leader abstraction *Omega*, the notion of 'last' is well-defined: there is a time after which an invocation will have the greatest round number.

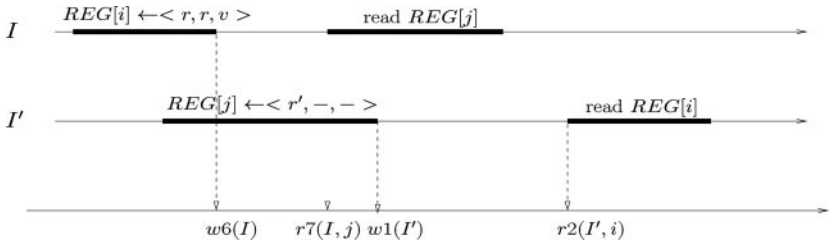

FIGURE 3. Regular register: read and write ordering.

property follows from this observation and the fact that only $\perp$ or a value deposited in a register *REG*[i] can be returned from an `Alpha()` invocation.

Conditional convergence Let I = `Alpha`$(r,-)$ be an invocation -by a process p_i—such that no invocation I' = `Alpha`$(r',-)$ with $r' > r$ starts before I returns. During the execution of I, we consequently have $\forall j \neq i$: $REG[j].lre < r$. It follows that I cannot be directed to return $\perp$ at line 3 or line 8. Moreover, the value determined by p_i at line 5 is necessarily a non- $\perp$ value. It follows from these observations that I returns at line 9 a non- $\perp$ value.

Quasi-agreement If none or a single invocation executes line 9, the quasi-agreement property is trivially satisfied. So, among all the invocations that return at line 9 (i.e. that return a non- $\perp$ value), let I = `Alpha`$(r,-)$ be the invocation with the smallest round number. Moreover, let I' = `Alpha`$(r', -)$ be any invocation such that $r' > r$ and that executes at least until line 6, i.e. an invocation that writes a value in $REG[1..n]$[8]. Let p_i (resp., p_j) be the process that invoked I(resp., I'), and v (resp., v') the value it writes in $REG[i]$ (resp., $REG[j]$). We show that $v' = v$, from which it follows that no two different values can be returned at line 9 (as a process that returns at line 9, returns the non-$\perp$ value it has just written in the array $REG[1..n]$).

We use the following time instant definitions (Figure 3).

- Definitions concerning I (I executes from line 1 until line 9):
  - Let $w6(I)$ be the time at which I terminates the write of the regular register $REG[i]$ at line 6. We then have $REG[i] = \langle r, r, v\rangle$.
  - Let $r7(I, j)$ be the time at which I starts reading $REG[j]$ at line 7. As p_i is sequential we have $w6(I) < r7(I, j)$.
- Definitions concerning I': (I' executes from line 6 until at least line 1, it can crash just after):
  - Let $w1(I')$ be the time at which I' terminates the write of the regular register $REG[j]$ at line 1. We then have $REG[j] = \langle r', -, -\rangle$.
  - Let $r2(I', i)$ be the time at which I' starts reading $REG[i]$ at line 2. As p_j is sequential we have $w1(I') < r2(I', i)$.

Let us first observe that, as I returns a non-$\perp$ value, it passed successfully the test of line 8, i.e. the value it read from $REG[j].lre$ was smaller than r. Moreover, when I' executed line 1, it assigned to $REG[j].lrer' > r$. As the register $REG[j]$ is regular we conclude that I started reading $REG[j]$ before I' finished writing it (otherwise, p_i would have read r' from $REG[j].lre$ and not a value smaller than r). Consequently, we have $r7(I, j) < w1(I')$, and by transitivity $w6(I) < r7(I, j) < w1(I') < r2(I', i)$. This is illustrated in Figure 3.

It follows that when I' reads $REG[i]$ at line 2, it obtains $\langle x, x,-\rangle$ with $x \geq r$ (this is because, after I, p_i has possibly executed other invocations with higher round numbers). Moreover, as I' does not return $\perp$ at line 12, when I' reads $REG[1..n]$ at line 2 it does not see a register $REG[k]$ such that $REG[k].lre > r'$. As we always have $REG[k].lre \geq REG[k].lrww$ for any register $REG[k]$, this means that, when I' determines a non-$\perp$ value v' at line 4, it obtains $v' = REG[k].val$ from some register $REG[k]$ such that $r' > REG[k].lre \geq REG[k].lrww \geq REG[i].lrww = x \geq r$. Let I'' be the invocation that has deposited v' in $REG[k].val$(I'' is consequently issued by p_k).

- If $REG[k].lrww = r$, we have $i = k$ (because r can be generated only by p_i—no two processes use the same round numbers), and consequently I'' is I. It follows that $v' = v$.
- Otherwise, I'' is not I. The invocation I'' by p_k has then written v' in $REG[k].val$ at line 6, with a corresponding round number r'' such that $r' > REG[k].lrww = r'' > r$. As, by assumption, I' and I'' can execute only the lines 1–6, we can replace in our reasoning I' by I'' and consider the pair of invocations (I, I'') instead of the pair (I, I').
- So, either I'' obtained v' written by I and then $v' = v$, or I'' obtains v' from another invocation I'''. We then consider the pair of invocations (I, I''') instead of the pair (I, I''), and so on. When considering the sequence of invocations defined by the round numbers, the number of invocations between I and I' is finite (there are at most $r' - r + 1$ invocations in this sequence). It follows that the chain of invocations conveying v' to I' is finite, and

[8] Let us remind that the ordering on the round numbers constitutes the basic intuition from which the algorithm is designed.

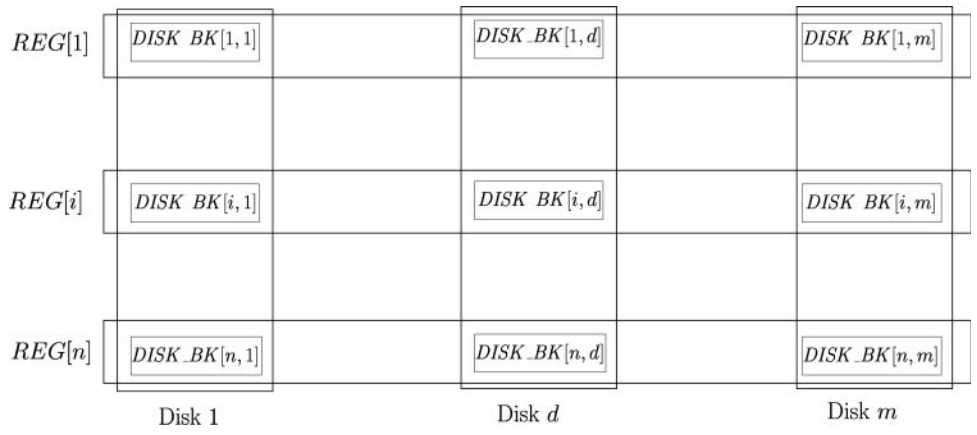

FIGURE 4. Replicating and distributing *REG*[1..*n*] on the *m* disks.

can only start with the invocation *I* that has written *v* in the corresponding register. It follows that $v' = v$.

Remark 1 It is important to notice that the only requirement that underlies the previous proof is the fact that the shared memory is reliable and provides 'regular register' semantics. There is no requirement on the number of processes that are allowed to crash.

Remark 2 The reader can observe that the test of line 3 is not used in the proof. This means that this line is not necessary for the algorithm correctness. Its aim is only to allow 'aborting' the current `Alpha`(*r*,−) invocation (directing it to return $\perp$) when it is known, from its very beginning, that it will return $\perp$. More generally, a copy of the lines 2–3 could be inserted at any place in the algorithm (e.g. between line 4 and line 5) without compromising its correctness.

5. IMPLEMENTING ALPHA WITH DISKS

Advances in hardware technology have made possible a new approach for storage sharing, where clients can access disks directly over a *storage area network*. The disks are directly attached to high speed networks that are accessible to clients. A client can access raw disk data (mediated by disk controllers with limited memory and CPU capabilities). These disks (usually called *commodity disks* or *network attached disks*) are cheaper than computers and are consequently attractive for achieving fault-tolerance [30]. This has motivated the design of disk-based consensus algorithms [8, 11].

5.1. Shared disks model

We consider here a 'shared memory' made up of *m* disks, each disk contains *n* blocks (one per process). We denote a disk by *d* and by *DISK_BK*[*i*, *d*] the block of the disk *d* associated with p_i (which means that only p_i can write this block while any process is allowed to read it). *DISK_BK*[−, *d*] denotes the overall set of blocks, see Figure 4.

A disk block *DISK_BK*[*i*, *d*] can be accessed by a read or a write operation. These operations are atomic: all read and write operations on a disk block can be totally ordered (there is no possibility of new/old inversion as allowed by regular registers). A disk can crash. When the disk *d* crashes, all its blocks *DISK_BK*[*i*, *d*] ($1 \leq i \leq n$) become inaccessible. After a disk becomes inaccessible, a read or a write operation on this disk either return $\perp$ or never return (it is then left pending forever). A disk that crashes in a run is *faulty* in that run; otherwise it is *correct*. It is assumed that a majority of disks are correct.

5.2. Underlying principle of the algorithm

When we consider the previous algorithm implementing a *Alpha* object, let us first observe that if each register *REG*[*i*] is replaced by a reliable disk, the algorithm still works. So, the idea consists in:

- First, we replicate each register *REG*[*i*] on each disk *d*($1 \leq d \leq m$) in order to make that register fault-tolerant (*REG*[*i*]; this is implemented by copies, namely the disk blocks DISK_BK [*i*, 1], . . . , *DISK*_BK [*i*, *m*]). So, each DISK_BK [*i*, *d*] is made up of three fields denoted *lrw*, *lrww* and *val* (with the same meaning as before) and initialized to $< 0 , -i, \perp >$.
- Then, we apply classic quorum-based replication techniques to the disks where a disk quorum is any majority set of disks. The assumption on the majority of correct disks guarantees that there is always a live quorum (this ensures termination), and two disk quorums always intersect (this ensures safety).

5.3. Building a reliable register from unreliable disks

The disk-based implementation of *Alpha* uses the well-known quorum-based replication technique to translate the read and

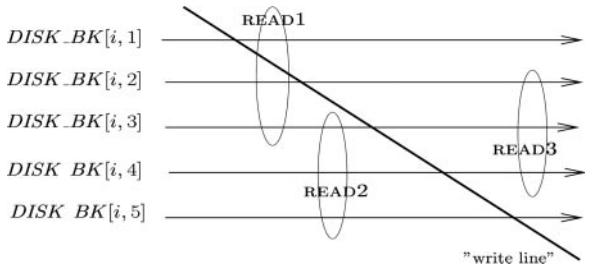

FIGURE 5. Implementing a 1WnR regular register with disks.

write operations on a 'virtual' object $REG[i]$ into its disk access counterparts on the m copies $DISK\_BK[i, 1], \ldots, DISK\_BK[i, m]$.

Each time it has to write new data into the virtual object $REG[i]$, p_i associates a new sequence number with that data, issues a write of that < pair on the corresponding block $DISK\_BK[i, d]$ of each of the m disks ($1 \leq d \leq m$). The write terminates when the pair has been written on a majority of disks. Similarly, a read of the virtual object $REG[i]$ is translated into m read operations, each one reading the corresponding block $DISK\_BK[i, d]$ on the disk d ($1 \leq d \leq m$). The read terminates when a pair has been received from a majority of the disks; the data with the greatest sequence number is then delivered as the result of the read of the virtual object $REG[i]$. Let us observe that, due to the 'majority of correct disks' assumption, every read or write operation on a virtual object $REG[i]$ always terminates.

It is easy to see that a read of the virtual object $REG[i]$ that is not concurrent with a write obtains the last data deposited in the object. For the read operations of $REG[i]$ that are concurrent with a write in $REG[i]$, let us consider Figure 5. That figure considers five disk blocks: $DISK\_BK[i, 1], \ldots, DISK\_BK[i, 5]$. A write of a virtual object $REG[i]$ by p_i is represented by a 'write line' the meaning of which is the following: the point where the 'write line' crosses the time line of a disk is the time at which that disk executes the write of the corresponding < data, sequence number > pair. As the system is asynchronous, these physical writes can occur as indicated in the figure, whatever their invocation times. The figure considers also three reads of $REG[i]$: each is represented by an ellipse and obtains the ⟨data, sequence number⟩ pair of the disks contained in the corresponding ellipsis (e.g. READ2 obtains the current pairs of the disk blocks $DISK\_BK[i, 3]$, $DISK\_BK[i, 4]$, and $DISK\_BK[i, 5]$. As we can see, each read obtains pairs from a majority of disks (let us notice that this is the best that can be done as, from the invoking process point of view, all the other disks can have crashed). READ1 and READ2 are concurrent with the write of the virtual object $REG[i]$, and READ1 obtains the new data, while READ2 obtains the old data. This new/old inversion is consistent with the definition of a regular register.

Remark If we were interested in obtaining an atomic register instead of a regular register, before returning its result, a read operation should first write in a majority of disks the value it is about to return. (A read operation has to 'help' the future read operations.) This would also require the ability for a disk to compare sequence numbers, as then each block of a disk could be written by all the processes, and not only by a single process (see Section 7). When implementing a shared register from disks, the difference between an atomic register and a regular register is that the former requires every read operation to help other reads (namely writing the read value). This helping is not required by regularity. (End of remark.)

Let us finally notice that a write by a process p_i that crashes can leave the disk blocks $DISK\_BK[i, 1], \ldots, DISK\_BK[i, m]$ in a state where the pair that is written has not been written to a majority of disks. Considering that a write during which the corresponding process crashes never terminates, this remains consistent with the definition of a regular register. As shown by Figure 5, all future reads will then be concurrent with that write and each of them can consequently return the old or the new data of $REG[i]$.

5.4. The algorithm

Let us look at Figure 2. The fields $REG[i].lre$ and $REG[i].lrww$ of a virtual object $REG[i]$ play actually the role of sequence numbers, the first for the writes issued at line 1, the second for the writes issued at line 6. Consequently, we can use these values as sequence numbers.

On another side, as far as disk accesses are concerned, we can factor out the writing of $REG[i]$ at line 2 and the reading of $REG[1..n]$ at line 2. This means that we can issue for each disk d, the writing of $DISK\_BK[i, d]$ and the reading of $DISK\_BK[1, d], \ldots, DISK\_BK[n, d]$, and wait until these operations have been executed on a majority of disks. When considering Figure 2, the same factorization can be done for the writing of $REG[i]$ at line 6 and the reading of $REG[1..n]$ at line 7.

The resulting disk-based algorithm (inspired by the Disk Paxos algorithm presented in [11]) is described in Figure 6. The variable $reg[i]$ is a local variable where p_i stores the last value of $REG[i]$ (there is no $REG[1..n]$ array, the array-like notation $reg[i]$ is only used for notational convenience). The algorithm tolerates any number of process crashes ($t < n$), and up to $(m - 1)/2$ disk crashes. It is wait-free as the progress of a process does not depend on the progress of the other processes.

This algorithm can be easily improved. For instance, when at line 4, p_i receives a triple $block[j, d]$ such that $block[j, d].lre > r$, it can abort the current attempt and return $\perp$ without waiting for triples from a majority of disks. (The same improvement can be done at line 13.) We did not include such optimizations in our algorithm for our aim is rather to show how an algorithm suited to a new context can

```
function Alpha (r, v): % This is the text for p_i %

    % [Step 1] ------------------------------------------------------------
    % 1.1. p_i first makes public the date of its last attempt %
    % 1.2. and reads the shared disks to know the other processes progress %
(1) reg[i].lre ← r;
(2) concurrently for each disk d : 1 ≤ d ≤ m do
(3)                                issue write reg[i] into DISK_BK[i, d];
(4)                                issue read DISK_BK[1, d], ..., DISK_BK[m, d] end_for;
(5) wait until (this has been successfully done at a majority of disks);
(6) let read_blocks = the set of triples block[j, d]. < lre.lrww, value > that have been read;
    % 1.3: p_i aborts its attempt if another process started a higher round %
(7) if (∃ block[j, d] ∈ read_blocks : block[j, d].lre > r) then return (⊥) end_if;
    % [Step 2] ------------------------------------------------------------
    % Then p_i adopts the last value that has been deposited %
    % If there is no such value, it adopts its own value v %
(8) let value be block[j, d].val where j, d are such that block[j, d] ∈ read_blocks ∧
          ∀ k, d' : block[k, d'] ∈ read_blocks : block[j, d].lrww ≥ block[k, d'].lrww;
(9) if (value = ⊥) then value ← v end_if;
    % [Step 3] ------------------------------------------------------------
    % 3.1. p_i writes the value it has adopted (together with the current date) %
    % 3.2. and reads the shared disks to know the other processes progress %
(10) reg[i] ← (r, r, value);
(11) concurrently for each disk d : 1 ≤ d ≤ m do
(12)                                issue write reg[i] into DISK_BK[i, d];
(13)                                issue read DISK_BK[1, d], ..., DISK_BK[m, d] end_for;
(14) wait until (this has been successfully done at a majority of disks);
(15) let read blocks = the set of triples block[j, d]. < lre, −, − > that have been read;
    % 3.3. p_i aborts its attempt if another process started a higher round %
(16) if (∃ block[j, d] ∈ read_blocks : block[j, d].lre > r) then return (⊥) end_if;
    % [Step 4] ------------------------------------------------------------
    % Otherwise, value is the result of the Alpha abstraction: p_i returns it %
(17) return (value)
```

FIGURE 6. Implementing Alpha with shared disks.

be derived by applying simple transformations to an algorithm designed for another context.

6. IMPLEMENTING ALPHA IN A MESSAGE-PASSING SYSTEM

6.1. Message-passing model

This section considers the case where the underlying system is a message-passing distributed system. There is no shared memory made up of shared registers or shared disks. Each pair of processes p_i pj is connected by a bidirectional channel that allows each of them to send messages to the other. To send a message m to pj, p_i invokes 'send m to pj''. A message is received by pj when pj returns from executing `receive ()` that provides it with a message[9]. The communication channels are reliable in the following sense: they neither lose, duplicate, nor corrupt messages. This means that every message that has been sent is received by its destination process (unless the destination process has crashed). There is no assumption on the time it takes for a message to go from its sender to its destination. The communication system is *asynchronous*.

6.2. From disks to message-passing

Let us consider Figure 4 that displays the way the array $REG[1..n]$ is replicated and distributed on the m disks. In a message passing system made up of n processes (without shared disks) we can take $m = n$ and implement each disk on a separate process, e.g. process p_i hosting disk I.

In that way, we obtain a simple peer system, where each p_i is a peer that on one side plays a process role for its client, and on the other side plays the role of a disk accessed by the set of processes. Such a transformation provides a message-passing version of *Alpha*, where the 'majority of correct disks' assumption becomes accordingly the 'majority of correct processes' assumption, i.e. we need to have $t < n/2$.

When we look at the disk blocks implemented by a process p_i, namely, the blocks $DISK\_BK[1, i], \ldots, DISK\_BK[n, i]$ defining column I in Figure 4, and the way the previous algorithms use their values, we see that both algorithms are

[9] In the message-passing algorithm described in Figure 7, the `receive ()` operation is not explicitly used. It appears implicitly at lines 13 and 16 in the statement '`upon the reception of` *msg_tag*(*param*)' where *msg_tag* denotes the type associated with the received message and *param* denotes its value.

```
function Alpha (r, v): % This is the text for p_i %

    % Step 1 ------------------------------------------------------------
    % 1.1. p_i first makes public the date of its last attempt %
(1) disk.lre ← r; for each j do send write_round&read (r) to p_j end_for;
    % 1.2. and reads disk variables to know the other processes progress %
(2) wait until (ack_wr&read (disk_j) received from a majority of processes p_j);
(3) let read_disks = the set of triples disk_j. < lre.lrww, val >that have been received;
    % 1.3: p_i aborts its attempt if another process started a higher round %
(4) if (∃ disk_j ∈ read_disks such that disk_j.lre > r) then return (⊥) end_if;
    % Step 2 ------------------------------------------------------------
    % Then p_i adopts the last value that has been deposited %
    % If there is no such value, p_i adopts its own value v %
(5) let value be disk_j.val where disk_j is such that disk_j ∈ read_disks ∧
                                 ∀k : disk_k ∈ read_disks: disk_j.lrww ≥ disk_k.lrww;
(6) if (value = ⊥) then value ← v end_if;
    % Step 3 ------------------------------------------------------------
    % 3.1. p_i writes the value it has adopted (together with the current date) %
    % 3.2. and reads disk variables to know the other processes progress %
(7) disk ←< r, r, value >;
(8) for each j do send write_val&read (disk) to p_j end_for;
(9) wait until (ack_wval&read (lre_j) received from a majority of processes p_j);
(10) let read_lre = the set of round numbers lre_j that have been received;
    % 3.3: p_i aborts its attempt if another process started a higher round %
(11) if (∃ lre_j ∈ read_lre such that lre_j > r) then return (⊥) end_if;
    % Step 4 ------------------------------------------------------------
    % Otherwise, value is the result of the Alpha abstraction: p_i returns it %
(12) return (value)

================================================

(13) upon the reception of write_round&read (r) from p_j do
(14)         disk.lre ← max(disk.lre, r);
(15)         send ack_wr&read (disk) to p_j

(16) upon the reception of write_val&read (< r, r, v >) from p_j do
(17)         if ((r ≥ disk.lre) ∧ (r > disk.lrww)) then disk ←< r, r, v > end_if;
(18)         send ack_wval&read (disk.lre) to p_j
```

FIGURE 7. Implementing Alpha in a message-passing system.

interested only in the maximum value of *DISK_BK*[1, *i*].*lre*, ..., *DISK_BK*[*n*, *i*].*lre* (see lines 3 and 8 in Figure 2, and lines 7 and 16 in Figure 6). The same observation holds for the set of pairs < *DISK_BK*[*j*, *i*].*lrww*, *DISK_BK*[*j*, *i*].*val* >, $1 \leq j \leq n$.

It follows that it is possible to benefit from the way the sequence numbers are used to shrink the array of disk blocks implemented by p_i into a single local variable *disk* such that:

- *disk.lre* = max (*DISK_BK*[1, *i*].*lre*, ..., *DISK_BK*[*n*, *i*].*lre*), i.e., the last round with which a process entered the *Alpha* object,
- *disk.lrww* = max (*DISK_BK*[1, *i*].*lrww*, ..., *DISK_BK*[*n*, *i*].*lrww*, the last round during which a process deposited a value in *disk.val*,
- *disk.val* = *DISK_BK*[*j*, *i*].*val* such that ∀*k*: *DISK_BK*[*n*, *i*].*lrww*) ≥ *DISK_BK*[*k*, *i*].*lrww*.

6.3. The algorithm

The resulting algorithm is described in Figure 7. For each process p_i, it consists of two parts: a part implementing the local invocation of Alpha (v_i), and a part dedicated to the implementation of the associated 'reduced' disk *disk*. The algorithm uses two types of request messages.

- The type *write_round&read* is used (at line 1) to tag messages carrying (1) the new round number entered by p_i, plus (2) a request for p_i to know the last *lre* and *lrww* rounds entered by the other processes. The answers to

```
(1) operation write_round&read (r)=
(2)     AC_DISK.lre ← max(AC_DISK.lre, r);
(3)     return (AC_DISK)

(4) operation write_val&read (reg)=
(5)     if ( (reg.lre≥ AC_DISK.lre)  ∧ (reg.lrww > AC_DISK.lrww) )
(6)             then AC_DISK ← reg end_if;
(7)     return (AC_DISK.lre)
```

FIGURE 8. The primitives of an active disk.

these requests are typed *ack_wr&read* (they are sent at line 15). The *lre* and *lrww* round values will be used by p_i to check whether it has to abort its current attempt, and if it does not abort, to compute the value it has to adopt.

- The type *write_val&read* is used (at line 8) to tag a message carrying (1) the value p_i wants to write (virtually in *REG*[*i*]) with the associated round number and (2) a request for p_i to know the last round entered by the other processes. The answers to these requests are typed *ack_wval&read* and need to carry a single *lre* round number (they are sent at line 18).

Messages are processed atomically by a process. Similarly, each invocation of Alpha () by p_i can only be interrupted at line 2 or at line 9. Moreover, the local variable *disk* of each process p_i is initialized to $\langle 0, -i, \perp \rangle$. (As in the previous section, this algorithm can be improved. A process p_i can abort the current attempt and returns $\perp$ without waiting for triples from a majority of processes, when at line 2 or 9, it receives a triple $disk_j$ such that $disk_j.lre < r$.) The algorithm obtained from this methodological construction can be seen as a variant, where failures are restricted to process crashes, of the original Paxos algorithm described in 12. (The message-passing version of Paxos considers that a process can crash and later recover, and also allows for message losses and message duplicates.)

7. IMPLEMENTING ALPHA WITH ACTIVE DISKS

7.1. On the disk side

An *active disk* is a disk that can atomically execute some operations more sophisticated than a simple read or a write operation. An example of active disk is described in [31]. The disk provides its users with an atomic *create* operation that atomically creates a new file object and updates the corresponding directory object. A disk can also provide processes with the ability to read and modify it with the same operation.

We have seen in Section 6.2 that, in the algorithm described in Figure 7, each process p_i is a peer playing two roles: one role consists in implementing `Alpha()`, while the other role consists actually in implementing an active disk that can atomically execute the two 'operations' *write_round&read* () and *write_val&read* (). After having decoupled each peer p_i as indicated, we obtain n active disks (one per process): $disk[1], \ldots, disk[n]$. Moreover, when we consider the semantics of the object we want to implement (namely a Alpha object), it appears that are relevant only:

- The value $disk[j].lre$ such that $\forall\ k : disk[j].lre \geq disk[k].lre$.
- The pair $< disk[j].lre,\ disk[k].val >$ such that $\forall\ k'$: $disk[k].lrww \geq disk[k'].lrww$.

This means that the whole array of disk blocks described in Figure 4 can be 'shrinked' to a single shared variable accessed by appropriate operations. This variable can be implemented by a reliable active disk (denoted *AC_DISK* and made up of three fields: *AC_DISK.lre*, *AC_DISK.lrww* and *AC_DISK.val*), and atomically accessed by two operations that (for consistency reasons) we call *write_round&read* () and *write_val&read* (). Both operations return values, their semantics is described in Figure 8. (These operations belong to family of '*read_modify_write* ()' operations.)

7.2. On the process side

The associated implementation of *Alpha* for a process p_i is described in Figure 9. It is a simple adaptation of the algorithm described in Figure 7. *ac_disk* is a local variable containing the last value of the active disk as read by p_i; r' is an auxiliary local variable used to contain a round number.

Interestingly, the resulting algorithm (that has been systematically constructed from a base message-passing algorithm) can be seen as a variant of an algorithm presented in [8]. It enjoys the same properties as algorithm based on shared registers described in Figure 2, namely, it tolerates any number of process crashes and is consequently wait-free.

7.3. Unreliable active disks

The previous algorithm assumes that the underlying active disk is reliable. An interesting issue is to build a reliable virtual active disk from unreliable base active disks. Unreliability means here that an active disk can crash (it does not corrupt its values): after it has crashed a disk does no longer executes the operations that are applied to it, before

```
function Alpha (r, v): % This is the text for p_i %

    % [Step 1] ------------------------------------------------------------
    % 1.1. p_i first makes public the date of its last attempt %
    % 1.2. and reads the active disk to know the other processes progress %
(1) ac_disk ← write_round&read (r) ;
    % 1.3: p_i aborts its attempt if another process started a higher round %
(2) if (ac_disk.lre > r) then return (⊥) end_if;
    % [Step 2] ------------------------------------------------------------
    % Then p_i adopts the last value that has been deposited %
    % If there is no such value, it adopts its own value v %
(3) if (ac_disk.val ≠ ⊥) then value ← ac_disk.val else value ← v end_if;
    % [Step 3] ------------------------------------------------------------
    % 3.1. p_i writes the value it has adopted (together with the current date) %
    % 3.2. and reads the active disk to know the other processes progress %
(4) r' ← write_val&read (< r, r, value >);
    % 3.3: p_i aborts its attempt if another process started a higher round %
(5) if (r' > r) then return (⊥) end_if;
    % [Step 4] ------------------------------------------------------------
    % Otherwise, value is the result of the Alpha abstraction: p_i returns it %
(6) return (value)
```

FIGURE 9. Alpha with an active disk.

crashing it executes them atomically. As before (Section 6), a disk that crashes during a run is said to be *faulty* with respect to that run; otherwise it is *correct*. Let us assume there are m active disks.

A simple way to build a correct virtual active disk consists in using the replication quorum-based strategy employed in Section 6. A process p_i issues *write_round&read* ()(or *write_val&read*()) operations on all the disks and waits until the operations have been successfully executed on a majority of active disks. Then, among all the triples of values returned by these invocations, p_i computes the maximal *lre* value, and the pair $\langle lrww, val\rangle$ such that *lrww* is maximal and considers the resulting triple as the result of the invocation of *write_round&read* () on the virtual active disk. The *write_val&read* () operation on the virtual active disk is implemented similarly.

The algorithm we obtain tolerates any number of process crashes ($t < n$), and up to $(m-1)/2$ disk crashes. It is wait-free as the progress of a process does not depend on the progress of the other processes.

8. CONSENSUS WITH INFINITELY MANY PROCESSES

Let us first observe that the specification of the leader abstraction *Omega* given in Section 2.3 does not involve process identities. It is consequently suited to work with an arbitrary number of processes. So, in order to get a consensus algorithm that works with infinitely many processes, we need to

- Ensure that no two processes use the same round numbers.
- Provide an implementation of *Alpha* that is not based on the partitioning of the shared memory into blocks such that, prior to the execution, each block is statically assigned to some process, namely the only process that can write it (as it is done in the algorithms described in the Figures 2, 6 and 7 where process identities need to be a *priori* known).

The Alpha object When we look at the algorithms based on a reliable active disk (Figures 8 and 9), we observe that there is no association between memory blocks and processes, and the number of processes is not used. It follows that these algorithms work whatever the identities and the number of processes (this number can be unknown or even unlimited [8]).

Private round numbers The *Alpha* abstraction assumes that distinct processes use distinct round numbers. If processes have different identities (e.g. integers), it is easy for them to forge distinct round numbers. An algorithm to 'name the anonymous' is described in [7].

A round number made up of ⟨sequence number, process id⟩ pair can then be associated with each invocation of `Alpha ()`. No two invocations have the same round number, and all the invocations can be totally ordered using a lexicographical order: $(sn1, i) < (sn2, j)$ if $((sn1 < sn2) \vee (sn1 = sn2 \wedge i < j))$. Lamport's logical clocks can generate 'close' sequence numbers [16]. (This can allow the leader to catch up quicker a higher round.)

It follows from the previous discussion that the consensus algorithm described in Figure 1 can be adapted to work with infinitely many processes by identifying each round with a ⟨sequence number, process id⟩ pair, and instantiating it with an implementation of *Alpha* based on active disks.

This conveys an interesting tradeoff relating the power of active disks with respect to commodity disks, and the information that has *a priori* to be known on the number and the identities of processes. Active disks are shared objects that provide all the processes with the same operations. Although they are more sophisticated than simple read/write operations, their implementation does not require partitioning the disk into one block per process. In contrast, a commodity disk provides the processes with a weaker read/write semantics, but each block of a commodity disk has to be associated with a single process (the only one that can write it).

9. CONCLUSION

Understanding the basic principles underlying the design of algorithms solving fundamental distributed computing problems, and finding the deep structure that unifies several algorithms solving similar problems, are challenging tasks at the heart of computer science.

This paper focuses on the basic principles that underlie the design of indulgent consensus algorithms. These algorithms are particularly robust as they tolerate the failure of the processes together with the 'instability' of the network. They clearly decouple the safety and liveness of consensus as advocated in Lamport's seminal Paxos algorithm [12].

Rather than introducing new indulgent consensus algorithms, each specific to a given model, the aim of this paper was to exhibit a simple abstraction, namely *Alpha*, which factors out the essence of the safety of indulgent consensus algorithms. Such an approach has allowed a systematic and incremental visit of several communication models, while providing each of them with a simple implementation of the proposed *Alpha* abstraction. More explicitly, starting from the shared memory model, each implementation has been obtained from the previous one by taking into account the peculiarities of the new target communication medium considered.

ACKNOWLEDGEMENTS

We are very grateful to the anonymous reviewers for their constructive comments that significantly helped improve the presentation of the paper.

REFERENCES

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

[2] Guerraoui, R. (2000) Indulgent algorithms. *In Proc. 19th ACM Symp. Principles of Distributed Computing*, Portland, OR, USA, July 16–19, pp. 289–298. ACM Press, New York.

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

[4] Chandra, T. and Toueg, S. (1996) Unreliable failure detectors for reliable distributed systems. *J. ACM*, **43**(2), 225–267.

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

[6] Raynal, M. (2005) A short introduction to failure detectors for asynchronous distributed systems. *ACM Sigact News, Distr. Comput. Column*, **36**(1), 53–70.

[7] Aguilera, M. K. (2004) A pleasant stroll through the land of infinitely many creatures. *ACM Sigact News, Distr. Comput. Column*, **35**(2), 36–59.

[8] Chockler, G. V. and Malkhi, D. (2002) Active disk paxos with infinitely many processes. In *Proc. 21th ACM Symp. Principles of Distributed Computing*, Monterey, CA, USA, July 21–24, pp. 78–87. ACM Press, New York.

[9] Merritt, M. and Taubenfeld, G. (2000) Computing with infinitely many processes. In *Proc. 14th Symp. Distributed Computing*, Toledo, Spain, pp. 164–178, *LNCS*, **1914**. Springer-Verlag, Berlin.

[10] Attiya, H. and Welch, J. (2004) *Distributed Computing, Fundamentals, Simulation and Advanced Topics* (2nd ed.). *Wiley Series on Parallel and Distributed Computing*. Wiley, New York.

[11] Gafni, E. and Lamport, L. (2003) Disk paxos. *Distr. Comput.*, **16**(1), 1–20.

[12] Lamport, L. (1998) The part-time parliament. *ACM Trans. Comput. Syst.*, **16**(2), 133–169, (A first version appeared as DEC Research Report, #49, September 1989).

[13] Boichat, R., Dutta, P., Frølund, S. and Guerraoui, R. (2003) Deconstructing paxos. *ACM Sigact News, Distr. Comput. Column*, **34**(1), 47–67.

[14] Guerraoui, R. and Raynal, M. (2004) The information structure of indulgent consensus. *IEEE Trans. Comput.*, **53**(4), 453–466.

[15] Boichat, R., Dutta, P., Frølund, S. and Guerraoui, R. (2003) Reconstructing paxos. *ACM Sigact News, Distr. Comput. Column*, **34**(2), 42–57.

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

[17] de Prisco, R., Lampson, B. W. and Lynch, N. A. (2000) Revisiting the paxos algorithm. *Theor. Comp. Sci.*, **243**(1–2), 35–91.

[18] Lamport, L. (2001) Paxos made simple. *ACM Sigact News, Dist. Comput. Column*, **32**(4), 34–58.

[19] Lampson, B. W. (1996) How to build a highly available system using consensus. In *Proc. 10th Int Workshop on Distributed Algorithms*, Bologna, Italy, October 9–11, pp. 1–17 in LNCS 1151. Springer-Verlag, Berlin.

[20] Abraham, I., Chockler, G. V., Keidar, I. and Malkhi, D. (2004) Byzantine disk paxos, optimal resilience with Byzantine shared memory. In *Proc. 23th ACM Symp. Principles of Distributed Computing*, St. John's, Newfoundland, Canada, July 22–25, pp. 226–235. ACM Press, New York.

[21] Guerraoui, R. and Raynal, M. (2006) A leader election protocol for eventually synchronous shared memory systems. In *Proc. 4th Int IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS'06)*, Gyeongju, Korea, April 26–28, pp. 75–80., IEEE Computer Society Press, Los Alamitos, CA.

[22] Dutta, P. and Guerraoui, R. (2002) Fast indulgent consensus with zero degradation. In *Proc. 4th European Dependable*

Computing Conf., Toulouse, France, October 23–25, pp. 191–208 *LNCS* **2485**, Springer-Verlag, Berlin.

[23] Mostéfaoui, A. and Raynal, M. (2001) Leader-based consensus. *Parallel Process. Lett.*, **11**(1), 95–107.

[24] Neiger, G. (1995) Failure detectors and the wait-free hierarchy. In *Proc. 14th ACM Symp. Principles of Distributed Computing*, Ottawa, Ontario, Canada, July 20–23, pp. 100–109, ACM Press, New York.

[25] Aguilera, M. K., Delporte-Gallet, C., Fauconnier, H. and Toueg, S. (2004) Communication-efficient leader election and consensus with limited link synchrony. In *Proc. 23th ACM Symp. on Principles of Distributed Computing*, St John's, Newfoundland, Canada, July 22–25, pp. 328–337. ACM Press, NewYork.

[26] Larrea, M., Fernández, A. and Arèvalo, S. (2000) Optimal implementation of the weakest failure detector for solving consensus. In *Proc. 19th Symp. Reliable Distributed Systems*, Nurnberg, Germany, October 16–18, pp. 52–60. IEEE Computer Society Press, Los Alamitos, CA.

[27] Mostéfaoui, A., Raynal, M. and Travers, C. (2004) Crash resilient time-free eventual leadership. In *Proc. 23th Symp. Reliable Distributed Systems*, Florianópolis, Brazil, October 18–20, pp. 208–218. IEEE Computer Society Press, Los Alamitos, CA.

[28] Mostéfaoui, A., Raynal, M. and Travers, C. (2006) Time-free and timer-based assumptions can be combined to get eventual leadership. *IEEE Trans. Parallel and Distr. Syst.*, **17**(7), 656–666.

[29] Lamport, L. (1986) On interprocess communication. *Distr. Comput.*, **1**(2), 77–101.

[30] Aguilera, M. K., Englert, B. and Gafni, E. (2003) On using network attached disks as shared memory. In *Proc. 21th ACM Symp. Principles of Distributed Computing*, Boston, MA, USA, July 13–16, pp. 315–324. ACM Press, New York.

[31] Gibson, G. A. *et al.* (1998) A cost-effective high-bandwidth storage architecture. In *Proc. 8th Int Conf. Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, USA, October 4–7, pp. 92–103. ACM Press, New York.

[32] Bernstein, P. A. and Goodman, N. (1981) Concurrency control in distributed data base systems. *ACM Comput. Survey*, **13**(2), 185–221.