

From a Store-Collect Object and Ω to Efficient Asynchronous Consensus

Michel Raynal^{1,2} and Julien Stainer²

¹ Institut Universitaire de France,

² IRISA, Université de Rennes 1, France

{raynal, Julien.Stainer}@irisa.fr

Abstract. This paper presents an efficient algorithm that build a consensus object. This algorithm is based on an Ω failure detector (to obtain consensus liveness) and a store-collect object (to maintain its safety). A store-collect object provides the processes with two operations, a store operation which allows the invoking process to deposit a new value while discarding the previous value it has deposited and a collect operation that returns to the invoking process a set of pairs (i, val) where val is the last value deposited by the process p_i . A store-collect object has no sequential specification.

While store-collect objects have been used as base objects to design wait-free constructions of more sophisticated objects (such as snapshot or renaming objects), as far as we know, they have not been explicitly used to built consensus objects. The proposed store-collect-based algorithm, which is round-based, has several noteworthy features. First it uses a single store-collect object (and not an object per round). Second, during a round, a process invokes at most once the store operation and the value val it deposits is a simple pair $\langle r, v \rangle$ where r is a round number and v a proposed value. Third, a process is directed to skip rounds according to its view of the current global state (thereby saving useless computation rounds). Finally, the algorithm benefits from the adaptive wait-free implementations that have been proposed for store-collect objects, namely, the number of shared memory accesses involved in a collect operation is $O(k)$ where k is the number of processes that have invoked the store operation. This makes the proposed algorithm particularly efficient and interesting for multiprocess programs made up of asynchronous crash-prone processes that run on top of multicore architectures.

Keywords: Asynchronous shared memory system, Building block, Concurrent object, Consensus, Distributed algorithm, Eventual leader, Failure detector, Fault-tolerance, Modularity, Multicore systems, Process crash, Store-collect object.

1 Introduction

1.1 On the Implementation of Consensus Objects

Consensus object and its universality An implementation of any object (or service) is *wait-free* if the crash of any number of processes does not prevent the other processes from terminating their operation invocations on the constructed object [13]. It has been shown by M. Herlihy [13] that *consensus objects* are universal when one has to design

wait-free implementation of any object (or service) defined from a sequential specification. This means that, as soon as we are provided with consensus objects and atomic read/write registers, it is possible to design algorithms (called *universal constructions*) that build wait-free implementations of any concurrent object defined by a sequential specification. Such implementations are said to be *linearizable* [14].

A *consensus* object is a one-shot object that provides the processes with a single operation denoted `propose()` (*one-shot* means that a process invokes at most once the operation `propose()` on a consensus object). When a process invokes `propose(v)`, we say that it “proposes *v*”. A consensus object allows processes to *agree* (if the processes of a multiprocess program do not have to agree in one way or another, they are independent and do not constitute a distributed computation). More specifically a consensus object is defined as follows. Each process is assumed to propose a value and has to decide a value in such a way that the following properties are satisfied: each correct process which invokes `propose()` decides a value (wait-free termination), a decided value is a proposed value (validity) and no two processes decide different values (agreement).

Consensus impossibility and ways to circumvent it. While consensus objects are fundamental objects for the design and the implementation of crash-prone distributed systems, the bad news is that they cannot be wait-free implemented in asynchronous systems. *Wait-free* means here “whatever the number of process crashes”. Moreover, this impossibility is independent of the underlying communication medium, which means that it holds for both read/write shared memory systems [18] and asynchronous message-passing systems [8]).

Several approaches to circumvent this impossibility have been investigated in the context of read/write shared memory systems. One consists in enriching the system model by providing the processes with registers stronger (from a computability point of view) than read/write atomic registers. This approach has given rise to the notion of *consensus number* introduced and developed by Herlihy [13]. An object *X* has consensus number *n* if *n* is the largest integer such that it is possible to wait-free implement *n*-process consensus objects from atomic read/write registers and objects *X*. If *X* allows to wait-free implement *n*-process consensus for any value of $n > 0$, the consensus number of the object *X* is $+\infty$. It is shown in [13] that there are objects (such as compare&swap or LL/SC registers) whose consensus number is $+\infty$.

Another approach consists in enriching the base read/write system with a failure detector [4]. Intuitively, a failure detector can be seen as a distributed module that provides each process with information on failures. According to the type and the quality of this information, several failure detectors can be defined. Failure detectors have initially been proposed for message-passing systems before being used in shared memory systems [17]. One of the most important result associated with the failure detector-based approach is the proof that the failure detector denoted Ω is the one that captures the minimal information on failures that allows processes to wait-free implement a consensus object despite asynchrony and process crashes [3]. A failure detector Ω is characterized by the following behavioral property: after a finite but unknown and arbitrary long period, Ω provides forever the processes with the same (non-crashed) leader.

Modular approach: on the liveness side. Implementations of consensus objects have to ensure that a single among the proposed values is decided (safety) and that each process

that proposes a value and does not crash eventually decides despite the behavior of the other processes (wait-freedom).

Interestingly, when considering round-based algorithms (i.e., algorithms in which the processes execute asynchronously a sequence of rounds), the safety and wait-freedom properties of a consensus object can be ensured by different means, i.e., by different object types. More precisely, the eventual leadership property provided by Ω can be used to ensure that at least one process will terminate (thereby entailing the termination of the other processes). Hence, an Ω failure detector constitutes a liveness building block on which implementations of consensus objects can rely in order to obtain the wait-freedom property.

Modular approach: on the safety side. To our knowledge three types of read/write-based objects that ensure the safety properties of a consensus object have been proposed.

The first (which is given the name *alpha* in [12,21]) has been proposed by Lamport in [16] in the context of message-passing systems and adapted to the read/write shared memory model by Lamport & Gafni [10]. An alpha object is a round-based abstraction which has a single operation denoted `deposit()`. *Round-based* means that the specification of `deposit()` involves the round number which is passed as a parameter. This operation takes also a value as input parameter and returns a proposed value or a default value \perp indicating that the current invocation is aborted. An alpha object is implemented with an array of n shared single-writer/multi-reader registers where n is the total number of processes. Each register contains two round numbers plus a proposed value or the default value \perp .

The second object, denoted *adopt-commit* has been introduced by Gafni [9]. It is a round-free object: its specification does not depend on round numbers, and its implementation requires two arrays of size n asynchronously accessed by each process one after the other. As *adopt-commit* objects are round-free, each round of an *adopt-commit*-based consensus algorithm requires its own *adopt-commit* object and, when it executes a round, a process accesses only the corresponding *adopt-commit* object.

A third object that has used to ensure the safety property of a consensus object is the *weak set* object type proposed by Delporte & Fauconnier [5]. This object is a set from which values are never withdrawn. Similarly to *adopt-commit* objects, these sets are round-free objects but, differently from them, during each round a process is required to access three distinct sets (the ones associated to the previous, the current and the next rounds).

1.2 Content of the Paper

Step complexity and number of objects. The step complexity (number of shared memory accesses) involved by each invocation of an operation on an alpha, *adopt-commit* or weak set object is $O(n)$.

On another side, the consensus algorithms based on *adopt-commit* or weak set objects requires one such object per round and, due the very nature of Ω , the number of rounds that have to be executed before processes decide is finite but can be arbitrary large. This means that the number of *adopt-commit* or weak set objects used in an

execution cannot be bounded and, consequently, these objects have to be dynamically created. (Due to the distributed nature of the computation and the possibility of failures, such dynamic object creations are much more difficult to manage than iterative or recursive objects creation in sequential or parallel failure-free computing.) Interestingly, alpha-based consensus algorithms (e.g., [10,11,16]) needs a single alpha object.

A question. Hence, the question: Is it possible to design a consensus object from Ω (for the consensus liveness part) and (for the consensus safety part) an object such that (a) a single instance of this object is necessary (as in alpha-based consensus) and (b) whose step complexity of each operation is *adaptive* (i.e. depends on the number of processes that have invoked operations and not on the total number of processes)? The paper answers positively the previous question. To that end it considers *store-collect* objects.

Store-collect object. Such an object, which can be seen as an array with an entry per process, provides processes with two operations denoted `store()` and `collect()`. The first operation allows a process to deposit a new value in the store-collect object, this new value overwriting the value it has previously deposited. The second operation is an asynchronous read of the last values deposited by each process. A *store-collect* object has no sequential specification.

While a *store-collect* object has a trivial wait-free implementation based on an array of size n with operations whose step complexity is $O(n)$, in a very interesting way, efficient adaptive wait-free implementations have been proposed. As an example, when considering the implementation described in [2], the step complexity of each invocation of `collect()` is $O(k)$ where k , $1 \leq k \leq n$, is the number of processes that have previously invoked the operation `store()` and the step complexity of each invocation of `store()` by a process is $O(1)$ but for its first invocation which can be up to $O(k)$.

A variant of a store-collect object is one in which the operations `store()` and `collect()` are merged to obtain a single operation denoted `store_collect()` (whose effect is similar to `store()` followed by `collect()`). A wait-free implementation of such a variant is described in [6] where it is shown that in some concurrency patterns the step complexity of `store_collect()` is $O(1)$.

Content of the paper. The paper presents an algorithm that wait-free implements a consensus object from Ω (building block for wait-free termination) and a single store-collect object (building block for consensus safety).

When compared to consensus algorithms that needs an unbounded number of adopt-commit or weak set objects, the proposed algorithm (similarly to alpha-based consensus algorithms [10,12,16]) needs a single base (store-collect) object. Moreover, when compared to alpha-based algorithms, it has several noteworthy features. (a) A better step complexity (measured as the number of accesses to the shared memory) during each round. (b) The fact that an entry of the store-collect object has only two components (a round number plus a proposed value) while an entry of an alpha object has three components (two round numbers plus a proposed value). And (c) the fact that the next round executed by a process is dynamically computed from its current view of the global state and not a priori defined from a predetermined sequence (thereby allowing a process to

skip useless computation rounds). It is important to notice that, while the algorithm is relatively simple, the proof that a single value is decided is not trivial.

Hence, the paper presents a new consensus algorithm suited to shared memory systems which, from an efficiency point of view, compares favorably with existing algorithms. It is important to notice that, with the advent of multicore architectures, the design of such efficient fault-tolerant algorithms become a real challenge.

Roadmap. The paper is made up of 4 sections. Section 2 presents the computation model (base read/write registers, store-collect object and Ω), and the consensus object. Then, Section 3 describes, discusses and proves correct an efficient algorithm that builds a consensus object from Ω and a single store-collect object as underlying building blocks. Finally, Section 4 concludes the paper.

2 Computation Model

2.1 Crash-Prone Asynchronous Processes

The system is made up of a set Π of n sequential processes denoted p_1, \dots, p_n . The integer i is the index of the process p_i . The processes are asynchronous which means that each process proceeds at its own speed which can vary arbitrarily. The execution of a sequential process is represented by a sequence of steps which are accesses to its local memory or to the shared memory (see below).

Any number of processes may crash. A crash is a premature halt. After it has crashed (if ever it does) a process executes no more step. It is only assumed that a process that does not crash eventually executes its next step as defined by the code of its algorithm. Given an execution, a process that crashes is said to be *faulty*, otherwise it is *correct*.

2.2 Cooperation Model

From a notational point of view, the names of the objects shared by the processes are denoted with upper case letters (e.g., *DEC*) while the name of a local variable of a process p_i is denoted with lower case letters with i as a subscript (e.g., set_i).

Cooperation objects: an atomic register and a store-collect object. The processes cooperate through an atomic multi-writer/multi-reader register denoted *DEC* (initialized to the default value \perp) and a single store-collect object denoted *MEM*. Such an object contains a set of pairs (i, v) where i is a process index and v a value. For any i , this set contains at most one pair $(i, -)$. Initially, a store-collect object is empty.

The operation store() and collect(). As indicated in the introduction, such an object has two operations denoted *store()* and *collect()*. A process p_i invokes *MEM.store(val)* to deposit the value val , i.e., the pair (i, val) is added to the store-collect and overwrites the previous pair $(i, -)$ (if any)¹. Hence, when (i, val) belongs to the store-collect object, val is the last value stored by the process p_i .

¹ In the algorithm proposed in Section 3 a value val is a pair made up of a round number r and a proposed value v . To prevent confusion, the notation $(-, -)$ is used for a pair written into a store-collect object, while the notation $\langle -, - \rangle$ is used for a pair val .

A process invokes $MEM.collect()$ to obtain a value of the store-collect object. The set that is returned is called a *view* and contains the latest pairs deposited by the processes that have invoked $MEM.store()$.

Partial order on the views. To define precisely the notion of “latest” pairs returned in a view, we use the following partial order relation on views. Let $view1$ and $view2$ be two views. The notation $view1 \leq view2$ means that, for every i such that $(i, v1) \in view1$, we have $(i, v2) \in view2$, where the invocation $MEM.store(v2)$ by p_i is issued after (or is) its invocation $MEM.store(v1)$.

Properties of the operations store() and collect() The invocations of these operations satisfy the following properties.

- **Validity.** Let col be an invocation of $collect()$ that returns the set $view$. For any $(i, v) \in view$, there is an invocation $store(v)$ issued by the process p_i that has started before the invocation col terminates.
This property means that a $collect()$ operation can neither read from the future, nor output values that have never been deposited.
- **Partial order consistency.** Let $col1$ and $col2$ be two invocations of the operation $collect()$ that return the views $view1$ and $view2$, respectively. If $col1$ terminates before $col2$ starts, then $view1 \leq view2$.
This property expresses the mutual consistency of non-concurrent invocations of the operation $collect()$: an invocation of $collect()$ cannot obtain values older than the values obtained by a previous invocation of $collect()$. On the contrary, there is no constraint on the views returned by concurrent invocations of $collect()$ (hence the name *partial order* for that consistency property).
- **Freshness.** Let st and col be invocations of $store(v)$ and $collect()$ issued by p_i and p_j , respectively, such that st has terminated before col starts. The view returned by p_j contains a pair (i, v') such that v' is v or a value deposited by p_i after v .
This property expresses the fact that the views returned by the invocations of $collect()$ are *up to date* in the sense that, as soon as a value has been deposited, it cannot be ignored by future invocations of $collect()$. If $store(v)$ is executed by a process p_i , the pair (i, v) must appear in a returned view (provided there are enough invocations of $collect()$) unless v has been overwritten by a more recent invocation of $store()$ issued by p_i .
- **Wait-free termination.** Any invocation of an operation by a process that does not crash terminates.

It is easy to see from these properties that a store-collect object has no sequential specification (two invocations of $collect()$ which obtain incomparable views cannot be ordered).

Wait-free implementations of store-collect objects. Such implementations are described in several papers (see Chapter 7 of [21] for a survey). The implementations described in [1,2] are based on atomic read/write registers. As noticed in the introduction, they are adaptive to the number k of processes that have invoked the operation $store()$. Let the *step complexity* of an operation be the maximum number of shared memory accesses it

can issue. When considering the implementation presented in [2], the step complexity of an invocation of `collect()` or of the first invocation of `store()` by a process is $O(k)$ and the step complexity of the other invocations of `store()` by the same process is $O(1)$.

Fast store-collect object Such an object, introduced in [6], is a store-collect object where the `store()` and the `collect()` operations are merged into a single operation denoted `store_collect()`. This object is particularly interesting when a process invokes repeatedly `store()` followed by `collect()` without executing other steps in between, which is exactly what the store-collect-based consensus algorithm presented in Section 3 does.

An implementation of such a store-collect object is presented in [6], where the step complexity of an invocation of `store_collect()` converges to $O(1)$ when, after some time, a single process invokes that operation².

2.3 The Failure Detector Ω

This failure detector, which has been informally defined in the Introduction, has been proposed and investigated in [3]. It provides each process p_i with a read-only variable denoted `leaderi` that always contains a process index. The set of these variables satisfies the following property.

- Eventual leadership. There is a finite time τ after which the local variables `leaderi` of all the correct processes contain the same process index and this index is the index of a correct process.

It is important to notice that, before time τ , there is an anarchy period during which the variables `leaderi` can have arbitrary values (e.g, there no common leader and crashed processes can be leaders). Moreover, τ can be arbitrarily large and is never explicitly known by the processes.

As already indicated, Ω is the weakest failure detector that allows a consensus object to be wait-free implemented [3]. Moreover, as consensus cannot be solved in a pure asynchronous read/write system prone to process crashes, it follows that such a system has to be enriched with time-related behavioral assumptions in order Ω can be built. Examples of such assumptions and associated Ω algorithms are described in [7].

Notation The previous read/write system model enriched with the additional computability power provided by Ω is denoted $ASM[\Omega]$.

3 The Store-Collect-Based Consensus Algorithm

This section presents and proves correct an algorithm that implements the operation `propose()` of a consensus object *CONS*. As previously announced, this construction is based on a store-collect object to ensure the consensus safety properties and a failure detector Ω to guarantee its wait-free termination property.

² As we will see, this is exactly what does occur in the proposed algorithm after Ω elects forever the same correct process.

3.1 Description of the Algorithm

Internal representation of the consensus object The two base objects used in the algorithm have been introduced in Section 2.2. The aim of the atomic register *DEC* is to contain the decided value. The aim of the store-collect object *MEM* is to guarantee that no two different values are decided.

The algorithm implementing the operation propose(). Algorithm 1 is a round-based asynchronous algorithm. A process p_i invokes $CONS.propose(v_i)$ where v_i is the value it proposes. Its invocation terminates when it executes the statement $return(DEC)$ where *DEC* contains the value it decides (line 17).

The local variable r_i contains the current round number of p_i while est_i contains its current estimate of the decision value (these local variables are initialized at line 1). A process executes a while loop (lines 2-16) until it decides (or crashes). Moreover, it executes the loop body (lines 4-14) only if it is currently considered as a leader by Ω (predicate of line 3).

When it is considered as a leader, p_i does the following. First it stores its current local state $\langle r_i, est_i \rangle$ into the store-collect object *MEM* and then reads its current content by invoking *MEM.collect()* (line 4). (Let us observe that line 4 can be replaced by the single statement $mem_i \leftarrow MEM.store_collect(\langle r_i, est_i \rangle)$ if one wants to use a fast store-collect object instead of a more general store-collect object.) Let us notice that line 4 is the only line where p_i accesses the store-collect object, i.e., the part of the shared memory related to the consensus safety property. All the other statements executed by p_i in a round (but the write into *DEC* if it decides) are local statements.

```

operation CONS.propose( $v_i$ ) is
(1)   $r_i \leftarrow 1; est_i \leftarrow v_i;$ 
(2)  while ( $DEC = \perp$ ) do
(3)    if ( $leader_i = i$ ) then
(4)      MEM.store( $\langle r_i, est_i \rangle$ );  $view_i \leftarrow MEM.collect()$ ;
(5)       $mem_i \leftarrow \{ \langle r, v \rangle \mid (-, \langle r, v \rangle) \in view_i \};$ 
(6)       $rmax_i \leftarrow \max\{r \mid \langle r, - \rangle \in mem_i\};$ 
(7)      if ( $r_i = rmax_i$ )
(8)        then  $set_i \leftarrow \{v \mid \langle r, v \rangle \in mem_i \text{ where } r \in \{rmax_i, rmax_i - 1\}\};$ 
(9)          if ( $r_i > 1$ )  $\wedge$  ( $set_i = \{est_i\}$ )
(10)           then  $DEC \leftarrow est_i$ 
(11)           else  $r_i \leftarrow r_i + 1$ 
(12)           end if
(13)        else  $est_i \leftarrow v$  such that  $\langle rmax_i, v \rangle \in mem_i$ ;  $r_i \leftarrow rmax_i$ 
(14)        end if
(15)      end if
(16)    end while;
(17)  return(DEC)
end operation.

```

Algorithm 1: The store/collect-based consensus operation *propose()*

Then, p_i stores into mem_i the pairs $\langle r, v \rangle$ contained in the view $view_i$ it has obtained (line 5) and computes the greatest round $rmax_i$ that, from its point of view, has ever been attained (line 6). Its behavior depends then on the fact that it is or not late with respect to $rmax_i$.

- If it is late ($r_i < rmax_i$), p_i jumps to the round $rmax_i$ and adopts as new estimate a value that is associated with $rmax_i$ in the view it has previously obtained (line 13).
- If it is “on time” from a round number point of view ($r_i = rmax_i$), p_i checks if it can write a value into DEC and decide. To that end, it executes lines 8-12. It first computes the set set_i of the values that are registered in the store-collect object with a round number equal to $rmax_i$ or $rmax_i - 1$, i.e., the values registered by the processes that (from p_i 's point of view) have attained one of the last two rounds. If p_i has passed the first round ($r_i > 1$) and its set set_i contains only the value kept in est_i , it writes it into DEC (line 10) just before deciding at line 17. If it cannot decide, p_i proceeds to the next round without modifying its estimate est_i (line 11).

Hence, the base principle on which rests this algorithm is pretty simple to state. (It is worth noticing that this principle is encountered in other algorithms that solve other problems such as termination detection of distributed computations). This principle can be stated as follows: processes execute asynchronous rounds (observation periods) until a process sees two consecutive rounds in which “nothing which is relevant has changed”.

3.2 Discussion

A particular case It is easy to see that, when all processes propose the same value, no process decides in more than two rounds whatever the pattern failure and the behavior of Ω . Similarly, only two rounds are needed when Ω elects a correct common leader from the very beginning. In that sense, the algorithm is optimal from a “round number” point of view [15].

On the management of round numbers In adopt-commit-based or alpha-based consensus algorithms, the processes that execute rounds do execute a predetermined sequence of rounds³. Differently, the proposed algorithm allows a process p_i that executes rounds to jump from its current round r_i to the round $rmax_i$ which can be arbitrarily large (line 13). These jumps make the algorithm more efficient. More specifically, let us consider a time τ of an execution such that (a) up to time τ , when a process executes line 9, the decision predicate is never satisfied, (b) processes have executed rounds and mr is the last round that has been attained at time τ , (c) from time τ , Ω elects the same correct leader p_ℓ at any process p_i , and (d) p_ℓ starts participating at time τ . It follows from the algorithm that p_ℓ executes the first round during which it updates r_ℓ to mr , and then (according to the values in the store-collect MEM) at most either the rounds mr and $mr + 1$ or the rounds mr , $mr + 1$ and $mr + 2$. As the sequence of rounds is not predetermined, p_ℓ saves at least $mr - 2$ rounds.

³ In an adopt-commit-based algorithm each process that executes rounds does execute the predetermined sequence of rounds numbered 1, 2, etc., while, in an alpha-based algorithm each process p_i that executes rounds does execute the predetermined sequence of rounds numbered i , $i + n$, $i + 2n$, etc.

3.3 Proof of the Algorithm

This proof is based only on the properties of Ω and the store-collect object MEM . It does not require MEM to be built from atomic registers (they can be regular registers only). Due to page limitation, the missing proofs can be found in [22].

Lemma 1. *If a process invokes first $MEM.store(\langle r, - \rangle)$ and later $MEM.store(\langle r', - \rangle)$, we have $r' > r$.*

Lemma 2. *Let $r > 1$. If a process p_i invokes $MEM.store(\langle r, v \rangle)$ at time τ , then there is a process p_j that has invoked $MEM.store(\langle r - 1, v \rangle)$ at a time $\tau' < \tau$.*

Lemma 3. *A decided value is a proposed value.*

Lemma 4. *No two processes decide different values.*

Proof. As a decided value is a value that has been written into DEC and a process writes at most once into DEC , the proof consists in showing that distinct processes do not write different values into DEC .

Preliminary definitions. Let $view_i^r$ be the value of $view_i$ obtained by p_i during round r . Let $\tau(i, r, b, st)$ and $\tau(i, r, e, st)$ be the time instants at which process p_i starts and terminates, respectively, the invocation of the operation $store()$ during round r . $\tau(i, r, b, cl)$ and $\tau(i, r, e, cl)$ have the same meaning when considering the invocation of the operation $collect()$.

Let r be the first round during which processes write into DEC , p_i one of these processes and v the value it writes. Let us observe that, due to line 9, $r > 1$; hence $r - 1$ exists. We claim that, for any w such that $(-, \langle r, w \rangle)$ is returned by an invocation of $collect()$ we have $w = v$ (Claim C1). It follows (a) from this claim that no process can decide a value different from v at round r and (b) from this claim, Lemma 1 and Lemma 2 that no process ever writes $\langle r', w \rangle$ with $r' > r$ and $w \neq v$. Consequently, no value different from v can be decided which proves the consensus agreement property.

Proof of the claim C1. Let w be any value such that $(-, \langle r, w \rangle)$ is returned by an invocation of $collect()$. To prove the claim (i.e., $w = v$), let us consider the following definition given for each value w .

1. Let $\tau(k_w, r_w, e, cl)$ be the first time instant at which a process (let p_{k_w} denote this process) returns from an invocation of $collect()$ (let r_w denote the corresponding round) and the view it obtains is such that $(-, \langle r, w \rangle) \in view_{k_w}^{r_w}$.
2. Let j_w be a process index such that $(j_w, \langle r, w \rangle) \in view_{k_w}^{r_w}$ (hence p_{j_w} invokes $store(\langle r, w \rangle)$).

We claim (Claim C2, proved in [22]) that (a) p_{j_w} executes round $r - 1$, and during that round both (b) invokes $store(\langle r - 1, w \rangle)$ and (c) executes line 11 (i.e., $r_{j_w} \leftarrow r_{j_w} + 1$).

To prove the claim C1, let us consider any process p_i that writes into DEC at round r (the first round during which processes write into DEC). This process obtained $view_i^r$ when it invoked $collect()$ at round r . Considering any value w and its associated process p_{j_w} as previously defined, we analyze the different cases which can occur according to value r' such that $(j_w, \langle r', v' \rangle) \in view_i^r$ or the fact that no pair $(j_w, -)$ belongs to $view_i^r$.

- $(j_w, \langle r', - \rangle)$ is such that $r' > r$. This case is not possible because otherwise we would have $rmax_i \geq r' > r$ when p_i executes round r and it would consequently execute line 13 and not line 10 (the line at which it writes into DEC).
- $(j_w, \langle r', v' \rangle)$ is such that $r' = r$. In that case, it follows from line 8 and the predicate evaluated by p_i at line 9 that we necessarily have $v' = v$. Moreover, as p_{j_w} writes at most once in a round (Lemma 1), it follows from the definition of j_w (see Item 2 above) that $v' = w$. Hence, $w = v$.
- $(j_w, \langle r', v' \rangle)$ is such that $r' = r - 1$. In that case, it follows from Item (b) of Claim C2 that p_{j_w} has invoked $store(\langle r - 1, w \rangle)$. Then the proof is the same as in the previous case, and we have $w = v$.
- $(j_w, \langle r', v' \rangle)$ is such that $r' < r - 1$ or there is no pair $(j_w, -)$ in $view_i^r$. It then follows from Item (a) of Claim C2 that p_{j_w} executes the round $r - 1$ and we have then $\tau(i, r, b, cl) < \tau(j_w, r - 1, e, st)$ (otherwise the freshness property of the store-collect object would be violated). According to the sequential code executed by p_{j_w} and p_i we consequently have

$$\tau(i, r, e, st) < \tau(i, r, b, cl) < \tau(j_w, r - 1, e, st) < \tau(j_w, r - 1, b, cl).$$

It then follows from the previous line, the freshness property of the store-collect object and the fact that p_i does not write the store-collect object after it has written into DEC , that $(i, \langle r, v \rangle) \in view_{j_w}^{r-1}$. Consequently, p_{j_w} reads $\langle r, - \rangle$ during round $r - 1$, it executes line 13 which contradicts Item (c) of Claim C2 (which states that p_{j_w} executes line 11 during round $r - 1$). Hence, this case cannot appear, which concludes the proof of Claim C1.

□ *Lemma 4*

Lemma 5. *Let assume that the eventual leader elected by Ω participates. Any correct process decides a value.*

The next theorem follows from Lemma 3, Lemma 4 and Lemma 5.

Theorem 1. *Let assume that the eventual leader elected by Ω participates. Algorithm 1 is a wait-free implementation of a consensus object in the system model $ASM[\Omega]$.*

4 Conclusion

This paper was motivated by the use of store-collect objects to build a consensus object. It has presented such an algorithm based on a single store-collect object in which a value stored by a process is a simple pair made up of a round number and a proposed value. Due to the fact that it uses a single store-collect object, the algorithm is practically interesting. Moreover, as it can benefit from the adaptive wait-free implementations that have been proposed for store-collect objects and it directs processes to skip rounds (thereby saving “useless” computation), this consensus algorithm is also particularly efficient and relevant for practical implementations. These features, together with its simplicity, make it attractive for multiprocess programs made up of asynchronous crash-prone processes that run on top of multicore architectures.

Acknowledgments. This work has been supported by the French ANR project DISPLEXITY devoted to computability and complexity in distributed computing.

References

1. Afek Y., Stupp G., Touitou D., Long-lived adaptive collect with applications. *Proc. 40th IEEE Symposium on Foundations of Computer Science Computing (FOCS'99)*, IEEE Computer Press, pp. 262-272, 1999.
2. Attiya H., Fouren A. and Gafni E., An adaptive collect algorithm with applications. *Distributed Computing*, 15(2):87-96, 2002.
3. Chandra T., Hadzilacos V. and Toueg S., The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685-722, 1996.
4. Chandra T. and Toueg S., Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225-267, 1996.
5. Delporte-Gallet C. and Fauconnier H., Two consensus algorithms with atomic registers and failure detector Ω . *Proc. 10th Int'l Conference on Distributed Computing and Networking (ICDCN'09)*, Springer Verlag #5408, pp. 251-262, 2009.
6. Englert B. and Gafni E., Fast collect in the absence of contention. *Proc. IEEE Int'l Conference on Distributed Computing Systems (ICDCS'02)*, IEEE Press, pp. 537-543, 2002.
7. Fernández A., Jiménez E., Raynal M. and Trédan G., A timing assumption and two t -resilient protocols for implementing an eventual leader service in asynchronous shared-memory systems. *Algorithmica*, 56(4):550-576, 2010.
8. 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.
9. Gafni E., Round-by-round fault detectors: unifying synchrony and asynchrony. *Proc. 17th ACM Symp. on Principles of Distr. Computing (PODC'98)*, ACM Press, pp. 143-152, 1998.
10. Gafni E. and Lamport L., Disk Paxos. *Distributed Computing*, 16(1):1-20, 2003.
11. Guerraoui R. and Raynal M., The information structure of indulgent consensus. *IEEE Transactions on Computers*. 53(4):453-466, 2004.
12. Guerraoui R. and Raynal M., The alpha of indulgent consensus. *The Computer Journal*, 50(1):53-67, 2007.
13. Herlihy M.P., Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124-149, 1991.
14. Herlihy M.P. and Wing J.L., Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463-492, 1990.
15. Keidar I. and Rajsbaum S., On the cost of fault-tolerant consensus when there are no faults. *ACM SIGACT News, Distributed Computing Column*, 32(2):45-63, 2001.
16. Lamport L., The part-time parliament. *ACM Trans. on Comp. Syst.*, 16(2):133-169, 1998.
17. Lo W.-K. and Hadzilacos V., Using failure detectors to solve consensus in asynchronous shared memory systems. *Proc. 8th Int'l Workshop on Distributed Algorithms (WDAG'94)*, Springer Verlag #857, pp. 280-295, 1994.
18. Loui M. and Abu-Amara H., Memory requirements for for agreement among Unreliable Asynchronous processes. *Advances in Computing Research*, 4:163-183, JAI Press Inc., 1987.
19. Mostéfaoui A. and Raynal M., Leader-based consensus. *Parallel Processing Letters*, 11(1):95-107, 2001.
20. Raynal M., Communication and agreement abstractions for fault-tolerant asynchronous distributed systems. *Morgan & Claypool Publishers*, 251 p., 2010 (ISBN 978-1-60845-293-4).
21. Raynal M., Concurrent programming: algorithms, principles and foundations. *To appear*, Springer, 420 pages, 2012.
22. Raynal M. and Stainer J., From a store-collect object and Ω to efficient asynchronous consensus. *Tech Report #1987*, 19 pages, IRISA/INRIA, Université de Rennes 1 (France), 2011.