An Efficient Distributed Obstruction-Free Compare-And-Swap Primitive

Pierre Sutra[1], Étienne Rivière[1], and Pascal Felber[1]

University of Neuchâtel, Switzerland

Abstract. Many modern distributed systems make use of an atomic read-modify-write primitive. Such a primitive is usually built on top of a central sequencer or relies on the Paxos consensus algorithm. These services are however inherently non-scalable, and as a consequence they constitute the bottleneck of the system. In this paper, we present a novel algorithm to implement a compare-and-swap primitive using a distributed shared atomic memory. Our algorithm is obstruction-free and implementable in a purely asynchronous manner. It is built on top of splitter and (weak) adopt-commit objects, as well as a novel shared abstraction named racing that we present in detail. To assess the benefits of our approach, we present a prototype implementation on top of the Cassandra data store, and we compare it empirically to the Zookeeper coordination service.

Contact author: Pierre Sutra, University of Neuchâtel, Emile-Argand 11, CH-2000 Neuchâtel, Switzerland, +41 32 718 2005, pierre.sutra@unine.ch.
Keywords: Asynchronous shared memory, consensus, compare-and-swap, state machine replication.

1 Introduction

Compare-and-swap is an important building block to implement non-blocking data structures and operations. In the message-passing world, however, such an abstraction is not part of the standard high-level API. More typically, message passing systems feature a consensus service that replaces it. This service is implemented via a sequencer or an agreement protocol (e.g., Paxos [16]).

Consensus and compare-and-swap have the same synchronization power [8]. However, distributed agreement protocols are known to be difficult to deploy and maintain. To address this issue, we investigate in this paper another approach where the agreement service is replaced with a compare-and-swap primitive implemented on top of a (distributed) asynchronous shared memory ($\mathcal{ASM}$). To make such a construction practical, this paper makes several contributions. First, we present an obstruction-free compare-and-swap object in $\mathcal{ASM}$ that works even if an unbounded number of processes access it. Second, we introduce a novel shared object, called racing, that encapsulates the behavior of algorithms that use one-shot objects to progress. We then explain how to construct compare-and-swap with a constant best-case time complexity on top of this abstraction. Our last contribution is an evaluation of a prototype implementation and a detailed performance comparison with the Zookeeper coordination service.

The outline of this paper is as follows. Section 2 presents our first construction of compare-and-swap. This construction works even if an unbounded number of processes access it. In Section 3, we introduce the notion of racing and implement on top of it a compare-and-swap primitive having a constant best-case time complexity. Section 4 compares a prototype implementation on top of Cassandra with the Zookeeper distributed lock service. We survey related work in Section 5, and concludes in Section 6. Due to space constraints, all the proofs are deferred to Appendix A.

2 Initial Construction

This section first presents our system model, and then describes several shared objects we make use of throughout this paper. At the end of this section, we depict a first construction of compare-and-swap. We shall refine this construction in the remainder of this paper.

2.1 System Model & Notations

We consider a classical asynchronous message-passing system characterized by a complete communication graph where both communication and computation are asynchronous. Processes take their identities from an unbounded set Π. During an execution, processes can fail-stop by crashing but we assume that at most f such failures occur.

Under the assumption that $|\Pi| > \lfloor \frac{f}{2} \rfloor$, the result of Attiya et al. [4] tells us that there exists an implementation of an asynchronous shared-memory ($\mathcal{ASM}$) in this model. Consequently, we shall write our algorithms in the $\mathcal{ASM}$ model where processes communicate by reading and writing to atomic multi-writer multi-reader (MWMR) registers. In what follows, we detail how to implement higher level abstractions from the shared registers.

All the objects we describe in this paper are linearizable [10]. An object is *one-shot* when a process may call one of its operations at most once. When there is no limit to the number of times a process may invoke the object's operations, the object is *long lived*.

In this paper, we are particularly interested in the following two liveness conditions on the invocations and responses of operations [9]: (*Obstruction-freedom*) if at some point in time a process runs solo then eventually it returns from the invocation; and (*Wait-freedom*) a process returns from the invocation after a bounded number of steps.

2.2 Weak Adopt-Commit

An adopt-commit object is a one-shot shared object that represents the agreement protocol of Gafni [6]. It can be used to implement round-based protocols such as consensus. An adopt-commit object is defined on a domain of values $\mathcal{V}$. Recently, Aspnes and Ellen [3] proved that, when $\mathcal{V}$ is unbounded, the solo time complexity of an adopt-commit object in $\mathcal{ASM}$ belongs to $\Omega\left(\sqrt{\log n}/\log\log n\right)$, with $n = |\Pi|$.

The first abstraction we employ to construct an efficient compare-and-swap primitive is a weak adopt-commit object. This object ensures the validity and coherence properties of an adopt-commit object, but weakens the convergence property to sidestep the lower bound on the time complexity. More precisely, a weak adopt-commit object is a shared object that exports a single operation $adoptCommit(u \in \mathcal{V})$ returning a tuple (f, v), with $f \in \{adopt, commit\}$ and $v \in \mathcal{V}$. During every history of a weak adopt-commit object, the following properties are satisfied:

(Validity.) If some process p adopts a value u then some process executed $adoptCommit(u)$ before.

(Coherence.) If some process p commits a value u then every process either adopts or commits u.

(Weak Convergence.) If no other process invoked *adoptCommit* before process p returns from $adoptCommit(u)$ then the invocation returns $(commit, u)$.

The rationale behind this modification is that we are going to construct an obstruction-free consensus on top of successive weak adopt-commit objects. The weak convergence property ensures that a process adopts its own value when it calls alone the object; this is sufficient to preserve obstruction-freedom.

Below, we present an implementation of a weak adopt-commit object. This algorithm can be seen as a variation of the wait-free consensus algorithm proposed by Luchangco et al. [17]. It makes use of a splitter object that detects a collision when two processes concurrently access the shared object. We first present the notion of splitter, then we detail our construction.

Algorithm 1 Weak Adopt-Commit – code at process p

```
 1: Shared Variables:
 2:     s                                   // A splitter object
 3:     c                                   // Initially, false
 4:     d                                   // Initially, ⊥
 5:
 6: adoptCommit(u) :=
 7:     if d ≠ ⊥ then
 8:         return (adopt, u)
 9:     if ¬ s.split() then
10:         c ← true
11:     if d = ⊥ then
12:         d ← u
13:     if c then
14:         return (adopt, d)
15:     return (commit, d)
16:
```

Splitter. The principle of a splitter was first introduced by Lamport [15]. It was later formalized by Moir and Anderson [18]. A splitter is a one-shot shared object that exposes a single operation: *split*(). This operation takes no parameter and returns a value in $\{true, false\}$.[1] When a process returns *true*, we shall say that it *wins* the splitter; otherwise it *loses* the splitter. When multiple processes call *split*(), at most one receives the value *true*. If now a single process calls *split*(), this call returns *true*. A splitter is implementable in a wait-free manner with atomic registers.

Algorithm 1 depicts a wait-free weak adopt-commit object. Our algorithm works as follows. Upon a call to $adoptCommit(u)$, a process p tries to win the splitter (line 9). If p fails, it raises the flag c to record that a collision occurs, i.e., the fact that two processes concurrently attempted to commit a value. Then, in case no decision was recorded ($d = \perp$ holds), process p writes the value it proposes into the shared register d (lines 11 and 12). The value returned from an invocation to *adoptCommit* is $(commit, d)$ in case c equals *false*, and $(adopt, d)$ otherwise.

2.3 Consensus

The second shared object we employ in our construction is an obstruction-free consensus object. Recall that consensus is a one-shot shared object whose interface consists of a single method *propose*(). This method takes as input a value from some set $\mathcal{V}$, and returns a value in $\mathcal{V}$ ensuring both

[1] A splitter is generally defined with returned values $\{L, S, R\}$. Here, we make no distinction between L and R.

Algorithm 2 Consensus – code at process p

```
 1: Shared Variables:
 2:     Obj                    // An unbounded array of weak adopt-commit objects
 3:     d                                                          // Initially, ⊥
 4:
 5: propose(u) :=
 6:     k ← 0
 7:     while true do
 8:         if d ≠ ⊥ then
 9:             return d
10:         k ← k + 1
11:         (f, u) ← Obj[k].adoptCommit(u)
12:         if f = commit then
13:             d ← u
14:
```

(*Validity*) if v is returned then some process invoked $propose(v)$ previously, and (*Agreement*) two processes always return the same value.

Algorithm 2 describes our obstruction-free implementation of consensus. This algorithm solely relies on weak adopt-commit objects. It can be seen as a variation of the construction proposed by Aspnes [2].

Algorithm 2 employs an unbounded array of weak adopt-commit objects: *Obj*. A process p that proposes a value u attempts to commit it by successively accessing the objects in the array. Every time p executes *adoptCommit* on some object $Obj[k]$, p updates its proposed value with the value returned by the object (line 11). In case the value returned by $Obj[k]$ is committed, p updates variable d with it (line 13). When p knows that d stored a value different than $\perp$, it returns this value as the result of the call to *propose* lines 8 and 9.

2.4 Compare-And-Swap

A compare-and-swap object exposes a single operation: $C\&S(u, v)$. This operation ensures that if the old value of the object equals u, it is replaced by v. In such a case the operation returns *true*; otherwise it returns *false*. We note $\top$ the initial value of a compare-and-swap object.

Algorithm 3 depicts an obstruction-free linearizable compare-and-swap algorithm in $\mathcal{ASM}$. It follows the pattern introduced by the universal construction of Herlihy [8] and the state machine replication approach [19], that is, processes agree via consensus on the next state of the shared object.

Algorithm 3 uses an unbounded shared array *Obj* of consensus objects. This array contains, for each $l \geq 0$, a consensus object $Obj[l]$ implemented with Algorithm 2. For some consensus object $C = Obj[l]$, we write $C.d$ to refer to the decision value of C. Initially, $Obj[0].d$ equals $\top$, and for every $l > 0$, $Obj[l].d$ equals $\perp$, i.e., the consensus object is not decided.

In detail, our algorithm works as follows. When executing $C\&S(u, v)$ a process p first determines the current value of the compare-and-swap object. This is done by retrieving $Obj[k]$, the last consensus object which was decided (line 9). If the value decided by $Obj[k]$ does not equal $(u, \_)$, where "_" means any value, the process p returns *false* (line 11). Otherwise, p tries replacing the state of the object with the pair (v, p). To that end, p proposes (v, p) to the consensus object $Obj[k + 1]$ (line 12). If process p wins the consensus, i.e., the consensus returns (v, p), process p returns *true* (line 13). Otherwise, p re-executes the previous code starting from (at least) $Obj[k + 1]$.

Algorithm 3 Compare-and-swap – code at process p

```
 1: Shared Variables:
 2:     Obj           // An unbounded array of consensus objects, with Obj[0].d = ⊤
 3:
 4: Local Variables:
 5:     k ∈ ℕ                                                      // Initially, O
 6:
 7: C&S(u, v) :=
 8:     while true do
 9:         k ← max (j ≥ k : Obj[j].d ≠ ⊥ ∧ Obj[j + 1] = ⊥)
10:         if Obj[k].d ≠ (u, _) then
11:             return false
12:         if Obj[k + 1].propose((v, p)) = (v, p) then
13:             return true
14:
```

Time Complexity. As expected with an obstruction-free object, there is no upper bound on the worst case time complexity of Algorithm 3. In addition, the result of Attiya et al. [5] tells us that the solo-finish time complexity, i.e., the number of step to terminate a call when the process runs solo in the worst case, belongs to $\Omega(n)$. Our implementation is in line with this impossibility result. Indeed, if $n - 1$ consensus objects were previously accessed, a process freshly calling $C\&S()$ may have to execute $O(n)$ steps to terminate (line 9 in Algorithm 3). However, notice that when a process accesses alone successively the object, a frequent case in many applications, the time complexity of an operation tends asymptotically toward $O(1)$. This comes from the fact that after one contention-free operation, i.e., an operation that does not have any concurrent operation, every successive contention-free operation in Algorithms 1, 2, and 3 has constant time complexity. To the best of our knowledge, we are the first to propose an algorithm having such a property without relying on a strong synchronization primitive.

3 Racing-based Construction

Algorithms 2 and 3 both use variable *Obj* to successively access weak adopt-commit and consensus objects. This section shows that the two usages are conceptually identical, and that we can capture them through a novel shared object named racing that we introduce in detail.

3.1 Notion of Racing

A *racing* on a domain $\mathcal{L}$, called laps, is a long-lived sequential object whose interface consists in operations $enter(p, l)$, and $leave(p, l)$. Given a history h of a racing object, we shall say that a process p *enters* (respectively *leaves*) lap l when $enter(p, l)$ (resp. $leave(p, l)$) occurs in h. A racing object assumes that a process always leaves the last lap it entered before it enters a new one. During every history h of a racing object, the following invariant holds:

(R1.) There exists a total order $\ll_h$ on the set of entered laps in h such that, for every process p that enters some lap l, either (i) some process left l before p enters it, or (ii) the last lap process p enters before l is the greatest lap smaller than l for the order $\ll_h$.

A racing object captures the semantics of variable *Obj* in Algorithms 2 and 3 within a single abstraction. In particular, we note that part (ii) of property R1 is always true for the two algorithms, and that in the case of Algorithm 2, part (i) never holds (a process cannot skip a lap). This last observation tells us that, as illustrated below, we may modify Algorithm 2 to use a loop pattern similar to Algorithm 3.

> *Construction 1:* Each time we execute a turn in the while loop (line 7), we execute the following code. Let o be the greatest decided weak adopt-commit object ($d \neq \perp$), or in case we already call it, an object higher in *Obj*. If o is decided, we update our proposal with its decision d. Otherwise, we invoke $o.adoptCommit(u)$.

In order to further illustrate this loop pattern, we depict in the following section a racing on $\mathbb{N}$. Then, we rewrite both Algorithms 2 and 3 on top of the racing abstraction.

3.2 Racing on $\mathbb{N}$

Algorithm 4 implements a racing on $\mathbb{N}$. This algorithm uses a shared mapping L from $\Pi \times \mathbb{N}$, and a variable c local to each process. For some process p, variables c and $L[p]$ store respectively the last lap entered by p and the last lap left by p. Initially, L is empty and c equals 0. In Algorithm 4, we write $L[x] \leftarrow y$ to add (x, y) to the map L, and the operation $codomain(L)$ returns the co-domain of L. Notice that such a shared mapping is implementable from MRMW registers in $\mathcal{ASM}$ (e.g., by employing a snapshot object [1]).

In detail, our algorithm works as follows. Upon calling $enter(p)$, process p computes the maximum element m in the co-domain of L ($m = 0$, in case L is empty). Then, p assigns c to $m + 1$, if $L[p] = m$ holds, and m otherwise (lines 9 to 12). The value of c is then returned. When p leaves lap c, it updates the value of $L[p]$. This assignment enforces invariant R1.

Algorithm 4 Racing on $\mathbb{N}$ – code at process p

```
 1: Shared Variables:
 2:     L                                   // An initialy empty mapping from Π to ℕ
 3:
 4: Local Variables:
 5:     c                                   // Initially, 0
 6:
 7: enter() :=
 8:     let m = max (codomain(L) ∪ {0})
 9:     if c = m then
10:         c ← m + 1
11:     else
12:         c ← m
13:     return c
14:
15: leave() :=
16:     L[p] ← c
17:
```

Consider again Algorithm 4 and assign to each natural k a one-shot object o returned each time a process enters lap k. Such a construction gives us a racing on one-shot objects, and it mimics the behavior of variable Obj in Algorithms 2 and 3. Based on this observation, the next section presents an implementation of consensus and compare-and-swap on top of a racing object.

Algorithm 5 Racing-based Consensus – code at process p

```
Shared Variables:
    R                                        // A racing on weak adopt-commit objects
    d                                        // Initially, ⊥

propose(u) :=
    while true do
        if d ≠ ⊥ then
            return d
        o ← R.enter()
        (f, u) ← o.adoptCommit(u)
        if f = commit then
            d ← u
        R.leave()
```

3.3 Application to Consensus and Compare-And-Swap

Algorithms 5 and 6 present our two implementations. Because these two algorithms employ a similar construction, we only delve into the details of our implementation of compare-and-swap in the following.

Algorithm 6 depicts an obstruction-free linearizable compare-and-swap primitive. This algorithm makes use of the following variables: R is a racing on consensus objects, variable C holds the latest consensus object entered locally, and s stores the latest state of the compare-and-swap object known locally. When a process p calls $C\&S(u,v)$, p first checks that the last consensus object it entered is decided (line 10). If it is the case, p updates its local variable s to reflect the state of the compare-and-swap object, and enter a new consensus object (line 13). Then, if s stores a value different than $(u,\_)$, p returns *false*. Otherwise, p proposes (v,p) to consensus, and if this call succeeds, p returns *true* (line 17). At this stage, if the operation invoked by p did not returned, process p re-executes the previous code.

Time Complexity. In a distributed system, because processes never add the same element to $domain(L)$ *concurrently*, the shared mapping of Algorithm 4 is implementable with a quorum system of $2f+1$ processes. The time complexity of such a construction (measured in message delay) is $O(1)$. Hence, we obtain a linearizable obstruction-free compare-and-swap having a best-case time complexity in $O(1)$. This algorithm works even if Π is unbounded. In the next section, to assess the practicability of our approach, we present a prototype implementation and compare its performance to the Apache Zookeeper coordination service.

Algorithm 6 Racing-based Compare-And-Swap – code at process p

Shared Variables:
    R // A racing on consensus objects

Local Variables:
    s // Initially, $\top$
    C // Initially, $R.enter()$

$C\&S(u, v) :=$
    while *true* **do**
        if $C.d \neq \bot$ **then**
            $s \leftarrow C.d$
            $R.leave()$
            $C \leftarrow R.enter()$
        if $s \neq (u, \_)$ **then**
            return *false*
        if $C.propose((v, p)) = (v, p)$ **then**
            return *true*

4 Empirical Assessment

To assess the practicability of our approach, we evaluate in this section a prototype implementation of Algorithm 6 that relies on the racing object described in Algorithm 4, and the consensus algorithm depicted in Algorithm 5. The implementation as well as the scripts we used for our experiments are publicly available [20]. Our prototype is built on top of the Apache Cassandra distributed data store [14]. Cassandra provides a distributed shared memory using consistent hashing and quorums of configurable sizes. Below, we describe the internals of our implementation, then we detail its performance in comparison to the Apache Zookeeper service [12]. This coordination service implements the Paxos consensus algorithm.

4.1 Implementation Details

Cassandra. The smallest unit of data in Cassandra is a column, a tuple that contains a name, a value, and a timestamp. Columns are grouped by rows. A column family contains a set of rows. This data model is close to the classical relational model used in databases. Each row is indexed by a key, and stored at a quorum of replicas (following a consistent hashing strategy). A client can read a row and write a column; the consistency of such operations is tunable. When the cluster running Cassandra is synchronized and Cassandra operates on quorums for both read and write operations, it provides an atomic snapshot model.

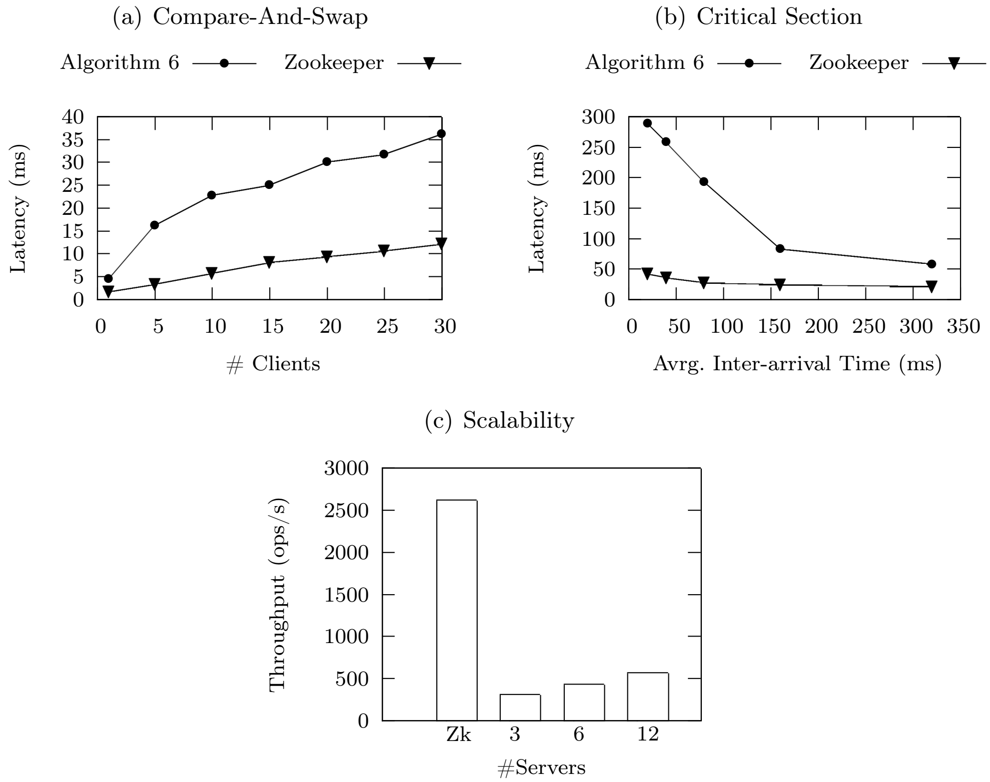

Fig. 1. Performance of our prototype implementation

Prototype Implementation. We implemented the different shared objects we described in the previous sections (splitter, weak adopt-commit, consensus, and compare-and-swap). Each object corresponds to a row in a column family named after the type of the object. When an object relies on lower-level abstractions, e.g., consensus employs multiple weak adopt-commit objects, the key of the lower abstractions is built by incrementing the key of the higher abstraction appropriately, then hashing this value. Our implementation uses the Python programming language. The conciseness of Python allows the whole implementation to be less then 500 lines of code.

4.2 Evaluation

Our experiments were conducted on a cluster of virtualized Xeon 2.5 Ghz machines running Ubuntu 12.04 GNU/Linux and connected by a 1 Gbps switched network. A *server machine* runs either Cassandra or Zookeeper. A *client machine* emulates multiple clients accessing concurrently the shared objects. During an experiment, a client executes 10^3 accesses to a shared object. In all the experiments, the client machine was not a bottleneck.

We first compare in Figure 1(a) the performance of our implementation to Zookeeper when the clients execute compare-and-swap operations. Zookeeper does not provide a compare-and-swap primitive. We devised the following simple implementation that relies on the versioning mechanism exposed to the clients by Zookeeper. When a client executes $C\&S(u, v)$ it first retrieves the value w and the attached version k of the znode uniquely identifying the object. In case $w = u$, the client writes v with version $k + 1$. If this write fails due to a concurrent writing, the client re-execute $C\&S(u, v)$. Notice that this implementation is not linearizable, because a read operation is executed by Zookeeper at a single server.

In Figure 1(a), we vary the number of clients accessing the same object. The initial state of the compare-and-swap is 0. Each client repeats an operation $C\&S(l, k)$ with k and l taken uniformly in $[\![0, 9]\!]$. We observe that when the number of client increases, both systems degrade. In case of Algorithm 6, this is due to the competition of the clients on the splitter objects. For Zookeeper, this comes from the fact that this service does not provide a full state machine replication abstraction. We thus had to implement $C\&S()$ at the client side. This inefficiency was recently pointed-out by Kalantari and Schiper [13].

In Figure 1(b), we compare the performance of each system when clients access a critical section (CS). Such an object is not in line with the non-blocking approach, but it is commonly used in distributed applications. In the case of Algorithm 6, the CS is guarded by a spinlock object implemented on top of compare-and-swap and using a back-off mechanism. We use the recipe describe in [12] for Zookeeper. We measured the average time a client takes to enter then leave the CS, and we vary the inter-arrival time of clients according to a Poisson distribution. We observe in Figure 1(b) that when the inter-arrival time is high, and thus little contention occurs, a client accesses the CS with Zookeeper in around 20 ms. For Algorithm 6, it takes around 60 ms, but the performance degrades quickly when clients access more frequently the CS. This comes from the fact that (1) we implemented a spinlock and thus clients are constantly accessing the system, and (2) as pointed out previously, when clients are competing on splitter objects, the performance of our algorithm degrades.

Figure 1(c) depicts the maximal throughput offered by our prototype implementation when clients execute successful operations on different objects (precisely, $C\&S(0, 0)$). With 6 and 12 servers we obtain respectively a multiplicative factor of 1.4 and 1.6. This scalability factor is small, nevertheless it shows that contrarily to the classical state machine replication approach, the system scales-up when more servers are added.

5 Related Work

A ratifier, or *adopt-commit*, object is a one-shot shared object that encapsulates the agreement and safety properties of consensus [2, 6]. In $\mathcal{ASM}$, when there is no assumption on the proposed values, the result of Aspnes and Ellen [3] proves that the number of steps to decide an adopt-commit object when executing solo belongs to $\Omega\left(\sqrt{\log n}/\log\log n\right)$. Luchangco et al. [17] present a consensus algorithm with a constant best-case time complexity, and a constant space-complexity. This algorithm assumes the existence of a compare-and-swap primitive, but solely access registers during an uncontended execution. The question of leveraging such a path in the implementation of an obstruction-free object was recently addressed by the work of Attiya et al. [5].

The time complexity of the universal construction of Herlihy [8] belongs to $O(n)$. Jayanti and Toueg [11] propose a variation of this construction that does not employ unbounded integers. In [5], the authors present a universal obstruction-free construction where operations are extended with pause and fail responses.

In a distributed system, the classical approach to implement consensus is the Paxos algorithm of Lamport [16]. The alpha abstraction of Guerraoui and Raynal [7] models the behavior of Paxos. It can be seen as a high-level view of successive consistent calls to adopt-commit objects. Our notion of racing introduced in Section 3 captures how processes successively access the compare-and-swap objects. We believe that such an abstraction helps to further understand the nature of consensus.

6 Conclusion

This paper investigates a novel approach to implement a distributed compare-and-swap primitive. Contrary to existing works, which mostly focus on state machine replication and the Paxos consensus algorithm [16], our approach employs only a distributed asynchronous shared memory. Hence, and as exemplified by our prototype, we can implement it in a client library that runs on top of an off-the-shelf distributed shared memory. This paper also shows that compare-and-swap is implementable with a constant best-case time complexity in $\mathcal{ASM}$, To obtain this result, we introduce weak adopt-commit and racing objects, two abstraction that we believe are of interest on their own.

Bibliography

[1] Afek, Y., Attiya, H., Dolev, D., Gafni, E., Merritt, M., Shavit, N.: Atomic snapshots of shared memory. J. ACM 40(4), 873–890 (Sep 1993)

[2] Aspnes, J.: A modular approach to shared-memory consensus, with applications to the probabilistic-write model. In: Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing. pp. 460–467. PODC '10, ACM, New York, NY, USA (2010)

[3] Aspnes, J., Ellen, F.: Tight bounds for anonymous adopt-commit objects. In: 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures. pp. 317–324 (Jun 2011)

[4] Attiya, H., Bar-Noy, A., Dolev, D.: Sharing memory robustly in message-passing systems. J. ACM 42(1), 124–142 (Jan 1995)

[5] Attiya, H., Guerraoui, R., Hendler, D., Kuznetsov, P.: The complexity of obstruction-free implementations. J. ACM 56(4), 24:1–24:33 (Jul 2009)

[6] Gafni, E.: Round-by-round fault detectors (extended abstract): unifying synchrony and asynchrony. In: Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing. pp. 143–152. PODC '98, ACM, New York, NY, USA (1998)

[7] Guerraoui, R., Raynal, M.: The alpha of indulgent consensus. Comput. J. 50(1), 53–67 (Jan 2007)

[8] Herlihy, M.: Wait-free synchronization. ACM Trans. Program. Lang. Syst. 13(1), 124–149 (Jan 1991)

[9] Herlihy, M., Shavit, N.: On the nature of progress. In: Anta, A.F., Lipari, G., Roy, M. (eds.) OPODIS. Lecture Notes in Computer Science, vol. 7109, pp. 313–328. Springer (2011)

[10] Herlihy, M., Wing, J.: Linearizability: a correcteness condition for concurrent objects. ACM Trans. on Prog. Lang. 12(3), 463–492 (Jul 1990)

[11] Jayanti, P., Toueg, S.: Some results on the impossibility, universality, and decidability of consensus. In: Segall, A., Zaks, S. (eds.) Distributed Algorithms, Lecture Notes in Computer Science, vol. 647, pp. 69–84. Springer Berlin Heidelberg (1992)

[12] Junqueira, F.P., Reed, B.C.: The life and times of a ZooKeeper. In: PODC '09: Proceedings of the 28th ACM symposium on Principles of distributed computing. pp. 4–4. ACM, New York, NY, USA (2009)

[13] Kalantari, B., Schiper, A.: Addressing the ZooKeeper synchronization inefficiency. In: Frey, D., Raynal, M., Sarkar, S., Shyamasundar, R.K., Sinha, P. (eds.) ICDCN. Lecture Notes in Computer Science, vol. 7730, pp. 434–438. Springer (2013)

[14] Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. SIGOPS Oper. Syst. Rev. 44(2) (Apr 2010)

[15] Lamport, L.: A fast mutual exclusion algorithm. ACM Trans. Comput. Syst. 5(1), 1–11 (Jan 1987)

[16] Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. 16(2), 133–169 (1998)

[17] Luchangco, V., Moir, M., Shavit, N.: On the uncontended complexity of consensus. In: Fich, F. (ed.) Distributed Computing, Lecture Notes in Computer Science, vol. 2848, pp. 45–59. Springer Berlin Heidelberg (2003)

[18] Moir, M., Anderson, J.: Fast, long-lived renaming. In: Tel, G., Vitányi, P. (eds.) Distributed Algorithms, Lecture Notes in Computer Science, vol. 857, pp. 141–155. Springer Berlin Heidelberg (1994)

[19] Schneider, F.B.: Implementing fault-tolerant services using the state machine approach: a tutorial. ACM Comput. Surv. 22(4), 299–319 (1990)

[20] Sutra, P.: http://github.com/otrack/pssolib (2013)

A Correctness of Algorithms 1 to 6

Theorem 1. *Algorithm 1 implements a wait-free linearizable weak-adopt commit object.*

Proof. Algorithm 1 employs only wait-free shared objects. Hence, to the light of its pseudo-code, Algorithm 1 keeps this liveness property. Both validity and weak convergence are immediate. To prove coherence now, consider that in some execution ρ of Algorithm 1, a process p commits a value u. Let *lin* be the linearization of ρ. Note t_0 the time in ρ where p executes line 13. Since p commits u, c always equals *false* before t_0. This implies that p wins the splitter s in *lin*. From the properties of a splitter object, we conclude that every process except p that executes line 11 in *lin* must execute line 10 before. This observation and the fact that $c = \mathit{false}$ at t_0 tell us that no process except p executes line 11 before t_0. Then, observe that because p executes line 11 before t_0, it executes line 12 at some time $t_1 < t_0$. As a consequence, d equals $\perp$ before time t_1, and d equals u after. This implies that the coherence property holds in ρ. □

Theorem 2. *Algorithm 2 implements an obstruction-free consensus.*

Proof. First of all, observe that a process that starts executing solo necessarily commits the value it proposes into the next weak adopt-commit object it enters alone. This is ensured by the weak convergence property of weak adopt-commit objects. As a consequence, Algorithm 2 is obstruction-free. The validity of consensus follows from the validity property of the weak adopt-commit objects used by the algorithm. It remains to prove that agreement is also satisfied. We first observe that if some process returns a value v, it necessarily accessed an object $\mathit{Obj}[k]$ that returned (commit, v). Let us name p the first process for which such a situation occurs. Since the execution of *adoptCommit* on $\mathit{Obj}[k]$ returns (commit, v) for process p (line 11), the coherence property of a weak adopt-commit object tells us that every process exiting object $\mathit{Obj}[k]$ must hold $u = v$. By a short induction, we know that if some process adopts or commits a value v' as the result of an access to $\mathit{Obj}[l \geq k]$, v' must equal v. Hence, agreement holds. □

Theorem 3. *Algorithm 3 implements an obstruction-free linearizable compare-and-swap object.*

Proof. Clearly, Algorithm 3 is obstruction-free. We prove now that the implementation is linearizable. Consider a history h produced by some execution ρ of Algorithm 3, and let o be some operation in h executed by a process p. Operation o is said *successful* iff it returns *true* in h. Consider the value of k at the time operation o returns. The consensus *associated* to o equals $Obj[k+1]$ if o is successful, and $Obj[k]$ otherwise. According to the code at lines 12 and 13, if $C\&S(u,v)$ is successful for p then denoting $Obj[k]$ its associated consensus, the agreement property of consensus implies that if at some point in time $Obj[k].d \neq \bot$ then it equals (v,p). Hence, for a consensus $Obj[k]$ there is at most one successful operation. We define lin as the serial ordering of the operations in h according to the order of their associated consensus in Obj, the successful operations being ordered before the non-successful operations having the same associated consensus. Consider now an operation $o = C\&S(u,v)$ by a process p in lin. When o is successful, the previous successful operation in lin is of the form $C\&S(\_,u)$. And when o is non-successful operations, the previous successful operation in lin is of the form $C\&S(\_,u')$ with $u' \neq u$, This reasoning tells us that lin forms a legal history. Finally, to ensure that real-time ordering holds in lin, observe that if o occurs after o' in h, it is linearized after o in lin because the consensus associated to o' follows necessarily the consensus associated to o in Obj. □

Theorem 4. *Algorithm 4 implements a wait-free linearizable racing with domain* $\mathbb{N}$.

Proof. Consider an execution ρ of Algorithm 4 in which the operations on the shared variable L are serialized, and note h the history produced by the folllowing mapping: When p invokes $enter()$ (resp. $leave()$) with $c = l$, it invokes $enter(p,l)$ (resp. $leave(p,l)$) in h. Similarly, when p returns from $enter()$ (resp. $leave()$) with $c = l$ in ρ, we add a response to $enter(p,l)$ (resp. $leave(p,l)$) in h. Note $(lin, <_{lin})$ the linearization of h induced by the order in ρ in which the operations on L occurs. Then, fix $\ll_h$ as the order on $\mathbb{N}$ reduced to the set of entered laps in h. Assume a process p enters some lap l in lin. We observe that p executes either line 10 or line 12. In the first case, this implies that p has just left $l-1$ which is the greatest lap smaller than l in which a process have entered during lin. In the second case, we observe that a process has left lap l, and moreover that this event occurs in lin before p enters l. Hence, invariant R1 holds in both cases. □

Theorem 5. *Algorithm 5 implements an obstruction-free consensus.*

Proof. Consider that a process p starts executing solo in Algorithm 5. Note o_{max} the greatest weak adopt-commit object entered so far by a process for the ordering $\ll_h$. By property R1 of a racing object, process p eventually enter a new object o at line 9, that is greater than o_{max} for the ordering $\ll_h$. By a short induction on the order $\ll_h$, it follows that no process entered in o before p. Hence, by the weak convergence property of every weak adopt-commit object, p commits its proposed value in o (line 10), and thus exits from the while loop at line 12. The validity of consensus follows immediately from the validity property of the weak adopt-commit objects used employed by Algorithm 5. To show agreement, consider that two processes p and q return from a call to *propose*() with two different values u and v. Let o and o' be the two adopt-commit object they employed respectively to commit their values. By the coherence property of a weak adopt-commit object, o must differ from o'. Assume then without lack of generality that o follows o' in the ordering $\ll_h$. By a short induction on the ordering $\ll_h$, we deduce that such a situation cannot occur. □

Theorem 6. *Algorithm 6 implements an obstruction-free linearizable compare-and-swap object.*

Proof. To prove obstruction-freedom, the reasoning is exactly identical to the one we depicted in Theorem 5. Then, to show that Algorithm 6 implements a linearizable compare-and-swap object, we apply the reasoning we used in Theorem 3, ordering in *lin* the successful operations according to $\ll_h$. □