CoVer-ability: **Co**nsistent **Ver**sioning for Concurrent Objects *

Nicolas Nicolaou † Antonio Fernández Anta † Chryssis Georgiou ‡

March 14, 2016

Abstract

An *object type* characterizes the domain space and the operations that can be invoked on an object of that type. In this paper we introduce a new property for concurrent objects, we call *coverability*, that aims to provide precise guarantees on the consistent evolution of an object. This new property is suitable for a variety of distributed objects including *concurrent file objects* that demand operations to manipulate the latest version of the object. We propose two levels of coverability: (i) strong coverability and (ii) weak coverability. Strong coverability requires that only a *single operation* can modify the latest version of the object, i.e. *"covers"* the latest version with a new version, imposing a total order on object modifications. Weak coverability relaxes the strong requirements of strong coverability and allows *multiple operations* to modify the same version of an object, where each modification leads to a different version. Weak coverability preserves consistent evolution of the object, by demanding any subsequent operation to only modify one of the newly introduced versions. Coverability combined with atomic guarantees yield to *coverable atomic read/write registers*. We also show that strongly coverable atomic registers are equivalent in power to consensus. Thus, we focus on *weakly* coverable registers, and we demonstrate their importance by showing that they cannot be implemented using similar types of registers, like ranked-registers. Furthermore we show that weakly coverable registers may be used to implement basic (weak) read-modify-write and file objects. Finally, we implement weakly coverable registers by modifying an existing MWMR atomic register implementation.

Submission Type: Regular paper.

*Supported in part by FP7-PEOPLE-2013-IEF grant ATOMICDFS No:629088, Ministerio de Economia y Competitividad grant TEC2014- 55713-R, Regional Government of Madrid (CM) grant Cloud4BigData (S2013/ICE-2894, co- funded by FSE & FEDER), NSF of China grant 61520106005, and European Commission H2020 grants ReCred and NOTRE.

†IMDEA Networks Institute, Madrid, Spain, nicolas.nicolaou@imdea.org,antonio.fernandez@imdea.org

‡Dept. of Computer Science, University of Cyprus, Nicosia, Cyprus, chryssis@cs.ucy.ac.cy

1 Introduction

Motivation and Prior Work. A concurrent system allows multiple processes to interact with a single object at the same time. A long string of research work [1, 5, 14–16] has been dedicated to explain the behavior of concurrent objects, defining the order and the outcomes of operations when those are invoked concurrently on the object. Lamport in [15,16] presented three different incremental semantics, *safety*, *regularity*, and *atomicity* that characterize the behavior of read/write objects (registers) when those are modified or read concurrently by multiple processes. The strongest, and most difficult to provide in a distributed system, is *atomicity* which provides the illusion that the register is accessed sequentially. Herlihy and Wing presented *linearizability* in [14], an extension of atomicity to general concurrent objects. More recent developments have proposed abortable operations in the event of concurrency [1], and ranked registers [5] that allow operations to abort in case a higher "ranked" operation was previously or concurrently executed in the system.

With the advent of cloud computing, emerging families of more complex concurrent objects, like files, distributed databases, and bulleting boards, demand precise guarantees on the consistent evolution of the object. For example, in *concurrent file objects* one would expect that if a write operation ω_2 is invoked after a write operation ω_1 is completed, then ω_2 modifies either the version of the file written by ω_1 or a version of the file newer than the one written by ω_1. *So is it possible to provide such guarantees using simpler objects as building blocks?*

In existing atomic read/write distributed shared register implementations, write operations are usually allowed to modify the value of the register, even when they are unaware of the value written by the latest preceding write operation. In systems that assume a single writer [2, 7, 11, 12], the problem may be diminished by having the sole writer compute the next value to be written in relation to the previous values it wrote. The problem becomes more apparent when multiple writers may alter the value of a single register concurrently [8, 19]. In such cases, atomic read/write register implementations appear unsuitable to directly implement objects that demand evolution guarantees. Closer candidates to build such objects are the bounded [3] and ranked [5] registers. These objects take into account the "rank" or sequence number of previous operations to decide whether to allow a read/write operation to commit or abort. These approaches do not prevent, however, the use of an arbitrarily higher rank, and thus an arbitrarily higher version, than the previous operations. This affects the consistent evolution of the object, as intermediate versions of the object maybe ignored.

Contributions. In this paper we propose a formalism to extend a concurrent object in such a way that the evolution of its state satisfies certain guarantees. To this end, we extend an object state with a *version,* and introduce the concept of *coverability,* that defines how the versions of an object can evolve (Section 3).

In particular, we first introduce a new class of a concurrent read/write register type, which we call *versioned register*. A concurrent register is of a *versioned* type, if the state of the register and any operation (read or write) that attempts to modify the state of the register, are associated with a *version*. An operation may modify the state and the version of the register, or it may just retrieve its state-version pair.

Coverability defines the exact guarantees that a versioned register provides when it is accessed concurrently by multiple processes with respect to the evolution of its versions. We define two levels of coverability: *strong* and *weak* coverability. Strong coverability ensures that only a *single operation* may change a given version (and thus the state) of the register, resulting in a lineal evolution of the versions (and the states) of the register. Weak coverability relaxes this rule and allows *multiple operations* to change a version, generating in this way a *tree* with possibly multiple version branches that can grow in parallel. This shares similarities with *fork linearizability* presented in [20]. However, in contrast to [20], weak coverability allows processes, that change the same version of the object, to see the changes of each other in subsequent operations. In particular, by weak coverability, when all the operations that extend a particular version of the object terminate, there is one version ver that was generated by one of those operations, which is the ancestor of any version extended by any subsequent operation. Thus, only a single branch in the tree is extended and that branch denotes the evolution of the register. Combining strong/weak coverability with atomic guarantees we obtain *strongly/weakly coverable atomic read/write registers*. While strongly coverable atomic registers are very desirable objects, we show that they are in fact very strong. In particular, we argue that these object types are as powerful as consensus objects (the details are given in Appendix B). Hence, it is challenging to implement these objects in some distributed

systems, and impossible in an asynchronous system prone to failures (from the FLP result [10]).

The good news is that even *weakly* coverable atomic registers have very interesting features. On the one hand, they can be implemented in message passing asynchronous distributed systems where processes can fail. To show this, we describe how algorithms that implement atomic R/W registers can easily be modified to implement these objects (Section 6). On the other hand, we show that weakly coverable atomic registers cannot be implemented using other previously defined register types such as ranked registers (Section 4).

One of the main motivation for introducing coverable registers are *file objects*, which can be seen as a special case of register objects in which each new value is a revision of the previous value. In essence, each modification of a file can be seen as an atomic read-modify-write (RMW) operation. Strongly coverable atomic registers provide the desired strong guarantees for files, since they are powerful enough to support atomic RMW operations. However, we show that even *weakly* coverable atomic registers can be used to provide interesting weak RMW guarantees that can be used to implement files with a good level of consistency (Section 5).

2 Model

We consider a distributed system composed of n *asynchronous* processes, with identifiers from a set $\mathcal{I} = \{p_1, \ldots, p_n\}$, each of which represents a sequential thread of control. Processes may interact with a set of shared objects $\mathcal{O}$. Each object in $\mathcal{O}$ represents a *data structure* shared among the processes, and has a *type* which defines the possible set of *object states* and the set of *operations* that provide the means to manipulate the object. A subset of processes may fail by *crashing*.

Processes can be modeled in terms of I/O Automata [18]. An automaton A (which combines the automata A_i for each process $p_i \in \mathcal{I}$) is defined over a set of *states* and a set of *actions*. An *execution* ξ of A is an alternating sequence of *states* and *actions* of A. An *execution fragment* is a finite prefix of an execution. We say that an execution fragment ξ' *extends* an execution fragment ξ, if ξ is a prefix of ξ'. A *history* of an automaton A, denoted by H_ξ, is the subsequence of actions occurring in some execution fragment ξ. An automaton A *invokes* an operation when an *invocation action* occurs in an execution ξ, and receives a *response* to an action when a *response action* occurs. An operation π is *complete* in an execution ξ, if H_ξ contains both the invocation and the matching response actions for π; otherwise π is *incomplete*. A history H_ξ of the automaton A_i of a process p_i is *well formed* if it begins with an invocation event and alternates between matching invocation and response events. (This demonstrates the assumption that each process is a single thread of control.) Each history H_ξ includes a precedence relation $\rightarrow_{H_\xi}$ on its operations. An operation π_1 *precedes* an operation π_2 (or π_2 *succeeds* π_1) in H_ξ if the response of π_1 appears before the invocation of π_2 in H_ξ. This is denoted by $\pi_1 \rightarrow_{H_\xi} \pi_2$. If $\pi_1 \not\rightarrow_{H_\xi} \pi_2$ and $\pi_2 \not\rightarrow_{H_\xi} \pi_1$ in H_ξ, then π_1 and π_2 are *concurrent*. A process p_i *crashes* in an execution ξ if the event fail_{p_i} appears and is the last action of p_i in H_ξ; otherwise p_i is *correct*.

3 Coverable Atomic Read/Write Registers

In this section we define a new type of R/W register, the *versioned register*. Next we provide new consistency properties for concurrent versioned registers called *(strong/weak) coverability*. We show how coverability can be combined with atomic guarantees to yield a coverable atomic register.

Versioned register. Let $Versions$ be a *totally ordered* set of *versions*. A *versioned register* is a type of read/write register where each value written is assigned with a version from the set $Versions$. Moreover, each write operation π that attempts to change the value of the register is also associated with a version, say ver_π, denoting that it intends to overwrite the value of the register associated with the version ver_π. More precisely, an implementation of a R/W register offers two operations: *read* and *write*. A process $p_i \in \mathcal{I}$ *invokes* a *write* (resp. *read*) operation when it issues a $\mathsf{write}(val)_{p_i}$ (resp. read_{p_i}) request. The *versioned* variant of a R/W register also offers two operations: (i) $\mathsf{cvr\text{-}write}(val, ver)_{p_i}$, and (ii) $\mathsf{cvr\text{-}read}()_{p_i}$. A process p_i invokes a $\mathsf{cvr\text{-}write}(val, ver)_{p_i}$ operation when it performs a write operation that attempts to change the value of the object. The operation returns the value of the object and its associated version, along with a flag

informing whether the operation has successfully changed the value of the object or failed. We say that a write is *successful* if it changes the value of the register; otherwise the write is *unsuccessful*. The read operation $\mathsf{cvr\text{-}read}()_{p_i}$ involves a request to retrieve the value of the object. The response of this operation is the value of the register together with the version of the object that this value is associated with.

Read operations do not incur any change on the value of the register, whereas write operations attempt to modify the value of the register. More formally, let Δ_T be the set of transitions for the versioned register. Then, each $\delta \in \Delta_T$ is a tuple $\langle \sigma, \pi, p_i, \sigma', res \rangle$, denoting that the register moves from state σ to state σ', and responds with res, as a result of operation π invoked by process $p_i \in \mathcal{I}$. The state of a versioned register is essentially its *value*, drawn from a set V, and its *version*, drawn from the set $Versions$. We assume that Δ_T is *total*, that is, for every $\pi \in \{\mathsf{cvr\text{-}write}(val, ver)_{p_i}, \mathsf{cvr\text{-}read}()_{p_i}\}$, $p_i \in \mathcal{I}$, and $\sigma = (val, ver) \in V \times Versions$, there exists $\sigma' = (val\prime, ver') \in V \times Versions$ and res such that $\langle \sigma, \pi, p_i, \sigma', res \rangle \in \Delta_T$. As such, the transitions of the versioned register type can be written as follows:

1. $\langle (val, ver), \mathsf{cvr\text{-}write}(val', ver_\omega), p_i, (val', ver'), (val', ver', chg) \rangle$, for $ver_\omega = ver$,
2. $\langle (val, ver), \mathsf{cvr\text{-}write}(val', ver_\omega), p_i, (val, ver), (val, ver, unchg) \rangle$, for $ver_\omega \neq ver$
3. $\langle (val, ver), \mathsf{cvr\text{-}read}(), p_i, (val, ver), (val, ver) \rangle$.

Notice that write operations may or may not modify the value/version of the register. In the transitions above, ver_ω denotes the version of the register which the write operation tries to modify. The relationship of ver with ver' may vary depending on the application that uses this register (but seems natural to assume that $ver' > ver$). A read operation does not make any changes on the value or the version of the object. To simplify notation, in the rest of the paper we avoid any reference to the value of the register. Additionally we only use the flag when its value is $unchg$. Thus, $\mathsf{cvr\text{-}write}(v, ver)(v, ver', chg)_{p_i}$ is denoted as $cvr\text{-}\omega(ver)[ver']_{p_i}$, and $\mathsf{cvr\text{-}write}(v, ver)(v', ver', unchg)_{p_i}$ is denoted as $cvr\text{-}\omega(ver)[ver', unchg]_{p_i}$.

We say that, a write operation *revises* a version ver of the versioned register to a version ver' (or *produces* ver') in an execution ξ, if $cvr\text{-}\omega(ver)[ver']_{p_i}$ completes in H_ξ. Let the set of *successful write* operations on a history H_ξ be defined as:

$$\mathcal{W}_{\xi,succ} = \{\pi : \pi = cvr\text{-}\omega(ver)[ver']_{p_i} \text{ completes in } H_\xi\}$$

The set now of produced versions in the history H_ξ is defined by:

$$Versions_\xi = \{ver_i : cvr\text{-}\omega(ver)[ver_i]_{p_i} \in \mathcal{W}_{\xi,succ}\} \cup \{ver_0\}$$

where ver_0 is the initial version of the object. Observe that the elements of $Versions_\xi$ are totally ordered. In the rest of the text we use '$*$' in the place of some parameter to denote that any legal value for that parameter can be used. Now we present the *validity* property which defines explicitly the set of executions that are considered to be valid executions.

Definition 1 (Validity) *An execution ξ (resp. its history H_ξ) is a* valid execution *(resp. history) on a versioned object, if $\mathcal{W}_\xi$ and for any $p_i, p_j \in \mathcal{I}$:*

- $\forall cvr\text{-}\omega(ver)[ver']_{p_i} \in \mathcal{W}_{\xi,succ}, ver < ver'$,
- *for any operations $cvr\text{-}\omega(*)[ver']_{p_i}$ and $cvr\text{-}\omega(*)[ver'']_{p_j}$ in $\mathcal{W}_{\xi,succ}$, $ver' \neq ver''$, and*
- *for each $ver_k \in Versions_\xi$ there is a sequence of versions $ver_0, ver_1, \ldots, ver_k$, such that $cvr\text{-}\omega(ver_i)[ver_{i+1}]$ $\in \mathcal{W}_{\xi,succ}$, for $0 \leq i < k$.*

Validity makes it clear that an operation changes the version of the object to a larger version, according to the total ordering of the versions. Also validity specifies that versions are *unique*, i.e. no two operations associate two states with the same version. This can be easily achieved by, for example, recording a counter and the id of the invoking process in the version of the object. Finally, validity requires that each version we reach in an execution is *derived* (through a chain of operations) from the initial version of the register ver_0. From this point onward we fix ξ to be a valid execution and H_ξ to be its valid history.

Coverability. We can now define the *strong* and *weak coverability* properties over a valid execution ξ of versioned registers with respect to some total order $>_\xi$ on the operations of ξ.

Definition 2 (Strong Coverability) *Let $ver_0 < ver_1 < \ldots < ver_{|\mathcal{W}_{\xi,succ}|}$ be the versions in $Versions_\xi$. A valid execution ξ is **strongly coverable** with respect to a total order $<_\xi$ on operations in $\mathcal{W}_{\xi,succ}$ if:*

- *cvr-$\omega(ver_{i-1})[ver_i] \in \mathcal{W}_{\xi,succ}$, for $1 \leq i \leq |\mathcal{W}_{\xi,succ}|$,*
- *cvr-$\omega(ver_{i-1})[ver_i] <_\xi$ cvr-$\omega(ver_i)[ver_{i+1}]$, for $1 \leq i < |\mathcal{W}_{\xi,succ}|$, and*
- *if $\pi_1, \pi_2 \in \mathcal{W}_{\xi,succ}$, and $\pi_1 \rightarrow_{H_\xi} \pi_2$ then $\pi_1 <_\xi \pi_2$.*

By Definition 2, all successful write operations are totally ordered with respect to the versions they modify. Notice than only a single write operation modifies each version ver_{i-1} to the next version ver_i. Thus, strong coverability defines an object type which is difficult to provide in an asynchronous distributed setting. In fact it can be shown that strongly coverable registers can be used to solve consensus among asynchronous fail-prone processes (see Appendix B). However, as shown by Fischer, Lynch and Paterson [10], solving consensus in such a system is impossible in the existence of a single crash failure, unless some powerful object is used. Hence the interest in defining a *weaker* version of coverability.

Definition 3 (Weak Coverability) *A valid execution ξ is **weakly coverable** with respect to a total order $<_\xi$ on operations in $\mathcal{W}_{\xi,succ}$ if:*

- *(**Consolidation**) If $\pi_1 =$ cvr-$\omega(*)[ver_i], \pi_2 =$ cvr-$\omega(ver_j)[*] \in \mathcal{W}_{\xi,succ}$, and $\pi_1 \rightarrow_{H_\xi} \pi_2$ in H_ξ, then $ver_i \leq ver_j$ and $\pi_1 <_\xi \pi_2$.*
- *(**Continuity**) if $\pi_2 =$ cvr-$\omega(ver)[ver_i] \in \mathcal{W}_{\xi,succ}$, then there exists $\pi_1 \in \mathcal{W}_{\xi,succ}$ s.t. $\pi_1 =$ cvr-$\omega(*)[ver]$ and $\pi_1 <_\xi \pi_2$, or $ver = ver_0$.*
- *(**Evolution**) let $ver, ver', ver'' \in Versions_\xi$. If there are sequences of versions $ver'_1, ver'_2, \ldots, ver'_k$ and $ver''_1, ver''_2, \ldots, ver''_\ell$, where $ver = ver'_1 = ver''_1$, $ver'_k = ver'$, and $ver''_\ell = ver''$ such that cvr-$\omega(ver'_i)[ver'_{i+1}] \in \mathcal{W}_{\xi,succ}$, for $1 \leq i < k$, and cvr-$\omega(ver''_i)[ver''_{i+1}] \in \mathcal{W}_{\xi,succ}$, for $1 \leq i < \ell$, and $k < \ell$, then $ver' < ver''$.*

By Definition 3, weak coverability allows multiple write operations to revise the same version ver_i of the register, each to a *unique* version ver_j. Given the set of successful operations $\mathcal{W}_{\xi,succ}$ and the set of versions $Versions_\xi$, Definitions 1 and 3 define a connected rooted tree $\mathcal{T}$ s.t.:

- The set of nodes of $\mathcal{T}$ is $Versions_\xi$,
- The initial version ver_0 of the object is the root of $\mathcal{T}$,
- A node ver_i is the parent of a node ver_j in $\mathcal{T}$ iff $\exists \pi(ver_i)[ver_j] \in \mathcal{W}_{\xi,succ}$,
- If $\pi_1 =$ cvr-$\omega(*)[ver_i] \in \mathcal{W}_{\xi,succ}$, s.t. π_1 is not concurrent with any other operation, then $\forall \pi_2 \in \mathcal{W}_{\xi,succ}$, s.t. $\pi_1 \rightarrow_\xi \pi_2$ and $\pi_2 = \pi(ver_z)[*]$, then ver_i is an ancestor of ver_z in $\mathcal{T}$, or $ver_i = ver_z$ (by Consolidation, Continuity, and Validity)
- if ver_i is an ancestor of ver_j in $\mathcal{T}$, then cvr-$\omega(*)[ver_i] <_\xi$ cvr-$\omega(*)[ver_j]$ (by Continuity).
- if ver_i is at level k of $\mathcal{T}$ and ver_j is at level ℓ of $\mathcal{T}$ s.t. $k < \ell$, then $ver_i < ver_j$ (by Evolution).

Observe that without the properties imposed by weak coverability, validity allows the creation of a tree of versions and does not prevent operations from being applied on an old version of the register. *Continuity*, *Consolidation*, and *Evolution* explicitly specify the conditions that reduce the branching of the generated tree, and in the case of not concurrency lead the operations to a single path on this tree. *Consolidation* specifies that write operations may revise the register with a version larger than any version modified by a preceding write operation, and may lead to a version newer than any version introduced by a preceding write operation.

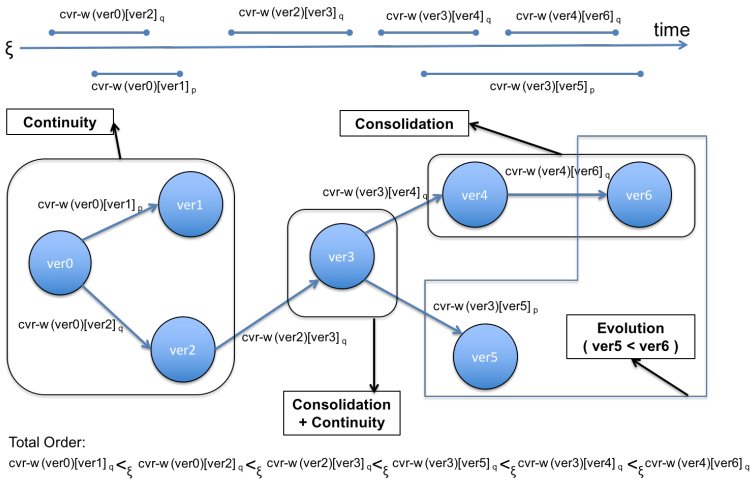

Figure 1: Tree Illustration from Weak Coverable Execution

Continuity defines that a write operation may revise a version that was introduced by a preceding write operation according to the given total order. Finally, *Evolution* limits the relative increment on the version of a register that can be introduced by any operation. Figure 1 provides an illustration of a tree created from a coverable execution ξ. We box sample instances of the execution and we indicate the coverability properties they satisfy.

Atomic coverability. We now combine coverability with atomic guarantees to obtain coverable atomic read/write registers. A register is linearizable [14], or equivalently *atomic* (as defined specifically for registers by [16, 17]) if the following conditions are satisfied by any execution ξ of an implementation of the object.

Definition 4 (Atomicity) [17, Section 13.4] An execution ξ of an automaton A is *atomic* if every *read* and *write* operation in ξ is *complete* and there is a partial ordering $\prec_{H_\xi}$ on all operations Π in H_ξ such that: **A1.** For any pair of operations $\pi_1, \pi_2 \in \Pi$, if $\pi_1 \rightarrow_{H_\xi} \pi_2$ then it cannot hold that $\pi_2 \prec_{H_\xi} \pi_1$, **A2.** If $\pi \in \Pi$ is a *write* operation and π' any operation in Π, then either $\pi \prec_{H_\xi} \pi'$ or $\pi' \prec_{H_\xi} \pi$, and **A3.** If v is the value returned by a *read* ρ then v is the value written by the last preceding *write* according to $\prec_{H_\xi}$ (or the initial value v_0 if there is no such a write).

In the context of versioned registers, in Definition 4, a *write* refers to a successful write $(cvr\text{-}\omega(*)[*, chg])$ operation on the versioned register. Therefore, all the write operations in an execution ξ are the ones that appear in $\mathcal{W}_{\xi,succ}$. A *read* refers to a versioned read $(cvr\text{-}\rho()[*])$ or an unsuccessful write $(cvr\text{-}\omega(*)[*, unchg])$ operation that does not modify the value (nor the version) of the register.

Definition 5 (Coverable atomic register) *A versioned register is* ***(strongly/weakly) coverable*** *and* ***atomic****, referred as* (strongly/weakly) coverable atomic register*, if any execution ξ on the register satisfies: (i) atomicity, and (ii) strong/weak coverability (Definition 2/3) with respect to the total order imposed by* **A2** *on $\mathcal{W}_{\xi,succ}$.*

Note that in a coverable atomic register, the ordering of read operations follows the ordering from atomicity. From this point onward, when clear from context, we refer to a coverable atomic register, as simply *coverable register*.

4 Weakly Coverable Atomic Registers vs Ranked Registers.

A type of registers that at first might resemble coverable registers are *ranked-registers* [5]. As we show here, ranked-registers are weaker than *weakly* coverable registers. In particular, we show that it is impossible to implement weakly coverable registers using ranked-registers; we begin by providing a formal definition of ranked-registers.

Definition 6 (Ranked-Registers [5]) *Let $Ranks$ be a totally ordered set of ranks with r_0 the initial rank. A ranked register is a MWMR shared object that offers the following operations: (i)* rr-read(r)*, with $r \in Ranks$ and returns $(r, v) \in Ranks \times Values$, and (ii)* rr-write$(\langle r, v\rangle)$*, with $(r, v) \in Ranks \times Values$ and returns $commit$ or $abort$. A ranked register satisfies the following properties: (i)* **Safety.** *Every* rr-read *operation returns a value and a rank that was written in some* rr-write *invocation or (r_0, v_0). Additionally, if $W =$* rr-write$(\langle r_1, v\rangle)$ *a write operation which commits and $R =$* rr-read(r_2) *such that $r_2 > r_1$, then R returns (r, v) where $r \geq r_1$. (ii)* **Non-Triviality.** *If a* rr-write *operation W invoked with a rank r_1 aborts, then there exists an operation with rank $r_2 > r_1$ which returns before W is invoked, or is concurrent with W (iii)* **Liveness.** *if an operation is invoked by a correct process then eventually it returns.*

We want to use rank-registers to implement the operations of a weakly coverable register. As in Section 2, we denote by $cvr\text{-}\omega(ver)[ver', flag]$ the coverable write operation that tries to revise version ver, and returns version ver' with a $flag \in \{chg, unchg\}$. Similarly we denote by $rr\text{-}\omega(r)[r_h, res]$ a write operation on a ranked-register that uses rank r and tries to modify the value of the register. The rank r_h is the highest rank observed by an operation and $res \in \{abort, commit\}$. In the following results we assume that a weakly coverable register is implemented using a set of ranked-registers. We begin with a lemma that shows that a coverable write operation revises the coverable register only if it invokes a write operation on some rank register and that write operation commits. *Omitted proofs can be found in Appendix A.*

Lemma 7 *Suppose there exists an algorithm A that implements a weakly coverable register using ranked-registers. In any execution ξ of A, if a process p_i invokes a coverable write operation $cvr\text{-}\omega(ver)[ver', chg]_{p_i}$, then p_i performs a write $rr\text{-}\omega(r)[r_h, commit]_{p_i,j}$ on some shared ranked-register j.*

Next we show that if π_1, π_2 are two non-concurrent write operations on the weakly coverable register, then π_2 performs a ranked write (that commits or aborts) on at least a single ranked register on which π_1 performed a committed ranked write operation. For the sake of the lemma R_i is the set of ranked registers on which π_i writes, and cR_i a subset of them on which the write commits.

Lemma 8 *Let $\pi_1 = cvr\text{-}\omega(ver)[ver_1, chg]_{p_i}$ and $\pi_2 = cvr\text{-}\omega(ver_1)[ver_2, *]_{p_z}, i \neq z$, be two write operations that appear in an execution ξ s.t. $\pi_1 \rightarrow_\xi \pi_2$. There exists some shared register $j \in R_2 \cap cR_1$ with a highest rank r_j before the invocation of π_1, such that p_i performs an $rr\text{-}\omega(r)[*, commit]_{p_i,j}$ during π_1, and p_z performs an $rr\text{-}\omega(r')[*, *]_{p_z,j}$ during π_2.*

Thus far we showed that a successful coverable write operation needs to commit on at least a single ranked register (Lemma 7), and two non-concurrent coverable write operations need to invoke a ranked write operation on a common rank register (Lemma 8). Using now Lemma 8 we can show that a coverable write operation that changes the version of the coverable register must use a rank higher than any previously successful coverable write operation.

Lemma 9 *In any execution ξ if $\pi_1 = cvr\text{-}\omega(ver)[ver_1, chg]_{p_i}$ and $\pi_2 = cvr\text{-}\omega(ver_1)[ver_2, chg]_{p_z}, z \neq i$, s.t. $\pi_1 \rightarrow_\xi \pi_2$, then there exists some shared register j such that p_i performs an $rr\text{-}\omega(r)[*, commit]_{p_i,j}$ during π_1, and p_z performs an $rr\text{-}\omega(r')[*, commit]_{p_z,j}$ during π_2, and $r' > r$.*

Now we prove our main result stating that a weakly coverable register cannot be implemented with ranked registers as those were defined in [5].

Theorem 10 *There is no algorithm that implements a weakly coverable register using a set of ranked registers.*

Proof. The theorem follows from Lemmas 7, 8, and 9, and the fact that a ranked register allows a write operation to commit even if it uses a rank smaller than the highest rank of the register. As by Lemma 7 a successful write must commit, then by ranked registers it can commit with a rank smaller than the highest rank of the accessed

register. This, however, by Lemma 9 may lead to violation of the consolidation and continuity properties and thus violation of weak coverability. □

Observe that the key fact that makes ranked registers weaker than weakly coverable registers is that the former allow write operations to commit even if their ranks are out of order. In particular, note that the Non-Triviality property *does not force* a write operation invoked with a rank r_1 to abort, even if there exists a completed prior operation with rank $r_2 > r_1$. As shown in [5] *non-fault-tolerant* ranked registers may preserve the total order of the ranks, and thus be used to implement consensus. As we show in Appendix B such ranked registers (i.e., that implement consensus) could be used to implement strongly coverable registers.

5 Applications of Weakly Coverable Atomic Read/Write Registers

Weak RMW registers. A shared object satisfies atomic *read-modify-write* (RMW) semantics if a process can atomically *read* and *modify* the value of the object using some function $\mathcal{F}$, and then *write* the new value on the object. Weakly coverable atomic R/W registers can be used to implement a weak version of RMW semantics. In a weak RMW object not all operations may successfully modify the value of the object. In case that a RMW operation is not concurrent with any other operation then this operation satisfies the RMW semantics. In case where two or more operations invoke RMW concurrently, at least one of them will satisfy the RMW semantics. Finally, weak RMW allow multiple RMW operations to modify successfully the same value.

Figure 2 presents an implementation of a weak RMW object using weakly coverable atomic R/W registers. We assume that the object offers a $\mathsf{rmw}(\mathcal{F})$ action that accepts a function and tries to apply that function on the value of the object. The object returns the initial value of the object and a flag indicating whether the value of the object was modified successfully.

At each process $i \in \mathcal{I}$
Local Variables: $lcver \in Versions, oldval, lcval, newv \in Values, flag \in \{chg, unchg\}$

function RMW($\mathcal{F}$)
  $\langle oldval, lcver\rangle \leftarrow$ cvr-read()
  $newv \leftarrow \mathcal{F}(oldval)$
  $\langle lcval, lcver, flag\rangle \leftarrow$ cvr-write$(lcver, newv)$
  if $flag == chg$ **then** return $\langle lcval, success\rangle$
  else return $\langle lcval, fail\rangle$

Figure 2: Weak RMW using Weakly Coverable Atomic R/W Registers

Theorem 11 *The construction in Figure 2 implements a weak RMW object.*

Proof. Consider an execution ξ of the algorithm. We begin the proof by studying the case where an operation $\mathsf{rmw}(\mathcal{F})$ is not concurrent with any other operation in ξ. The atomic nature of the register ensures that cvr-read returns the latest value and version, say $\langle ver, val\rangle$, written on the register. When the cvr-write operation is invoked, the write operation tries to modify the value associated with version ver. As there is no concurrent operation, the version of the register remains ver and thus according to *consolidation and continuity*, the write operation successfully writes the new value completing the RMW operation.

Consider now the case of two operations, π_1 and π_2, invoking rmw concurrently. Each of these operations involve a cvr-read followed by a cvr-write operation. Let ρ_{π_i} (resp. ω_{π_i}) denote the read (resp. write) operation invoked during π_i, for $i \in [1, 2]$. We have the following cases wrt the order of these operations: (i) $\omega_{\pi_1} \rightarrow \rho_{\pi_2}$, (ii) $\omega_{\pi_2} \rightarrow \rho_{\pi_1}$, (iii) $\rho_{\pi_2} \rightarrow \omega_{\pi_1} \rightarrow \omega_{\pi_2}$, (iv) $\rho_{\pi_1} \rightarrow \omega_{\pi_2} \rightarrow \omega_{\pi_1}$, or (v) ω_{π_1} is concurrent with ω_{π_2}. In case (i), both read and write operations of π_1 complete before the read and write operations of π_2 are invoked. In this case notice that the version of the object remains the same from the read to the write operation of both operations. Thus, according to *consolidation and continuity*, both write operations will successfully change the value of the register. The same holds for case (ii), where π_2's ops complete before the invocation of π_1's ops. In case (iii) the write operation of π_1 completes before the write operation of π_2. Let ρ_{π_2} in this case complete

At each process $i \in \mathcal{I}$
Local Variables: $lcver \in Versions$, initially ver_0 $lcval, newv \in Values$, initially $\perp$ $flag \in \{chg, unchg\}$, initially chg

function REVISE(v, ver)
  $\langle lcval, lcver, flag\rangle \leftarrow$ cvr-write(ver, v)
  if $flag == chg$ **then**
    return OK
  return $\langle lcval, lcver\rangle$

function GET()
  $\langle lcval, lcver\rangle \leftarrow$ cvr-read()
  return $\langle lcval, lcver\rangle$

Figure 3: File Object using Weakly Coverable Atomic R/W Registers

before ω_{π_1}. Both read operations ρ_{π_1} and ρ_{π_2} discover by *atomicity* the same version, say ver. So both write operations will be invoked as cvr-write(ver, v). Since no operation changes the version of the register before ω_{π_1} is invoked, then by *consolidation and continuity*, ω_{π_1} changes the version of the object to, say, ver_{π_1}. Notice that by *validity*, $ver_{\pi_1} > ver$. When ω_{π_2} is invoked it fails by *consolidation* to change the value of the object as $\omega_{\pi_1} \rightarrow \omega_{\pi_2}$ and it tries to change the version $ver < ver_{\pi_1}$ (the version of ω_{π_1}). Hence, only π_1 will manage to preserve RMW semantics. Similarly, we can show that only π_2 will preserve RMW semantics in case (iv). Finally, in case (v) if both writes try to change the version ver, both may succeed and preserve RMW semantics. Since, however, their versions are unique and comparable, then by *consolidation* any subsequent operation will RMW the highest of the two versions. So in all cases at least a single operation satisfies the RMW semantics, as desired. □

From the proof we can extract that weakly coverable registers may allow multiple writes to change the same version of the register, but *consolidation* ensures that at least one write satisfies RMW semantics for each version. Finally, *consolidation and continuity* ensure that eventually RMW operations diverge in a single path in the constructed tree.

Concurrent File Objects A file object can be implemented directly using RMW semantics since one can retrieve, revise, and write back the new version of the file. As RMW semantics can be used to solve consensus [13], they are impossible to be implemented in an asynchronous system with a single crash failure. Therefore, we consider file objects that comply to the weak RMW semantics as those were given in the paragraph above. In particular, we consider *concurrent file objects* that allow two fundamental operations, *revise* and *get* to be invoked concurrently by multiple processes. The revise operation is used to change the contents of the file object, whereas the get action is analogous to a read operation and facilitates the retrieval of the contents of the file. Semantically, a file object requires that a revise operation is applied on the latest version of the file and a get operation returns the file associated with the latest written version. Depending on the implementation, the values written and returned by these operations can be the complete file object, a fragment of the file object, or just the journal containing the operations to be applied on a file (similar to a journaled file system).

Figure 3 presets the algorithm that implements the two operations. The *revise* operation specifies the version of the file to be revised along with the new value of the shared object. The cvr-write operation attempts to perform the write with the given version and returns the value and version of the register, and whether the write succeeded or not. If the write succeeded then the operation informs the application for the proper completion of the revise operation; otherwise the latest discovered value-version pair is returned. From Theorem 11 and Figure 3 we may conclude the following theorem.

Theorem 12 *The construction in Figure 3 implements a file object.*

6 Implementing Weakly Coverable Atomic Read/Write Registers

We now show how we can implement weak coverable atomic registers. We do so by enhancing the Multi-Writer version of algorithm ABD [2, 19] (referred as MWABD) to preserve the properties of weak coverability. The presented technique can be applied to implementations of atomic R/W objects that utilize a $\langle tag, value\rangle$ pair to order the write operations and where each write performs two phases before completing: a *query phase* to obtain the latest value of the atomic object and a *propagation phase* to write the new value on the object. We

cvr-write($val, ver = maxtag$)

query-phase: Send query request to *all* the replicas and wait to receive $(tag, value)$ responses from a majority of them. Select the $(tag, value)$ among the collected replies with the largest tag; let the $\langle \tau, v \rangle$ be this pair and the integer component of τ be z. Then:

- If $ver == \tau$ then: Create a new tag $\tau_{new} = \langle z+1, wid \rangle$ where wid is the unique identifier of the writer and set $val_{new} = val$.
- If $ver \neq \tau$ then: Set $\langle \tau_{new}, val_{new} \rangle = \langle \tau, v \rangle$.

propagation-phase: Send $\langle \tau_{new}, val_{new} \rangle$ to all the replicas and wait to receive responses from a majority of them. if $ver == \tau$ then respond with $\langle val_{new}, \tau_{new}, chg \rangle$, otherwise respond with $\langle val_{new}, \tau_{new}, unchg \rangle$ to the process.

cvr-read()

query-phase: Send query request to *all* the replicas and wait to receive $(tag, value)$ responses from a majority of them. Select the $(tag, value)$ among the collected replies with the largest tag; let the $\langle \tau, v \rangle$ be this pair and the integer component of τ be z.

propagation-phase: Send $\langle \tau, v \rangle$ to all replicas and wait for responses from a majority of them. Respond with $\langle v, \tau \rangle$ to the process.

at-replica

On receipt of query message: Send the tag-value pair $\langle \tau_r, v_r \rangle$ stored locally.

On receipt of propagation message: Let $\langle \tau_m, v_m \rangle$ be the tag-value pair enclosed in the received message and $\langle \tau_r, v_r \rangle$ the local pair on the replica. Compare the tags τ_m and τ_r. If $\tau_m > \tau_r$ then store $\langle \tau_m, v_m \rangle$ locally. Reply with "ack".

Figure 4: The operations of algorithm vMWABD.

could also adopt implementations of stronger objects like the ones presented in [3–6] but we preferred to show the simplest modification in a fundamental algorithm. To capture the semantics of a coverable atomic register we modify the operations of algorithm MWABD to comply with the versioned variant of the R/W register. We use cvr-write(ver, v) and cvr-read() as the write and read operations respectively. A cvr-write(ver, v) operation may impact differently the state of the object, depending on the version of the shared object: it may appear as a *read* not modifying the value nor the version of the register or as a *write* changing both the value and the version of the register.

In brief, the original MWABD replicates an object to a set of hosts $\mathcal{S} \subset \mathcal{I}$ and it uses $\langle tag, value \rangle$ pairs to order the *read* and *write* operations. A tag consists of a *non-negative integer* number and a *writer identifier* which is used to break the ties among concurrent write operations. Both the read and write protocols have two phases: a *query* and a *propagation* phase. During the *query* phase the invoking process broadcasts a query message to all the replica hosts (replicas) and waits for a majority of them to reply with their tag-value pairs. Once those replies are received the process discovers the largest tag-value pair among the replies. In the second phase, a read operation propagates the discovered tag-value pair to the majority of the replicas. A write operation increments the largest tag, associates the new tag with the value to be written, and propagates the new tag-value pair to the majority of the replicas.

In the *versioned* MWABD, vMWABD for short, we use the tags associated with each value to denote the version of the register. The pseudocode of each operation of vMWABD is described in Figure 4. The cvr-read operation is similar to the read operation of MWABD with the difference that it returns both the value and the version of the register. A cvr-write operation differs from the original write by utilizing a condition before its *propagation* phase and depending whether the condition holds it changes the state of the register (value and version) or not, as detailed in Figure 4. Note that the version parameter of the write operation is equal to the maximum tag that the invoking process witnessed.

Theorem 13 *Algorithm* vMWABD implements weak coverable atomic registers.

Proof. It is clear that vMWABD still satisfies properties **A1-A3**. Any write operation that is not successful can be mapped to a read operation that performs two phases and propagates the latest value/version of the register to a majority of replicas before completing. It remains to show that vMWABD also satisfies the properties of validity and weak coverability.

Validity is satisfied since each tag is unique, as it is composed by an integer and the id of a process. The tag is monotonically incrementing at each replica, as according to the algorithm a replica updates its local copy only if a higher tag is received. A writer process discovers the maximum tag $maxtag$ among the replicas

and in the second phase it generates a tag $\langle maxtag + 1, wid \rangle$. As the tag at each replica is monotonically incrementing then each writer never generates the same tag twice. Also, for every write $cvr\text{-}\omega(tag)[tag', chg]$, $tag' = \langle tag.ts + 1, wid \rangle \Rightarrow tag' > tag$. Finally, since every tag is generated by extending the initial tag and each write operation extends a tag that obtains during its query phase then there is a sequence of tags leading from the initial tag to the tag used by the write operation.

For *consolidation* we need to show that for two write operations $\omega_1 = cvr\text{-}\omega(*)[tag_1, chg]$ and $\omega_2 = cvr\text{-}\omega(tag_2)[*, chg]$, if $\omega_1 \rightarrow_\xi \omega_2$ then $tag_1 \leq tag_2$. According to the algorithm ω_1 propagates tag_1 to the majority of replicas before completing. In the query phase, ω_2 receives messages from the majority of replicas. So there is one replica s that received tag_1 from ω_1 before replying to ω_2. Since the tag in s is monotonically incrementing, then s replies to ω_2 with a tag $tag_s \geq tag_1$. So ω_2 receives a $maxtag \geq tag_1$. Since ω_2 also changes the value and version of the register it means that its local tag tag_2 is equal to $maxtag$. This shows immediately that $tag_2 \geq tag_1$, completing the proof.

Continuity is preserved as a write operation first queries the replicas for the latest tag before proceeding to the propagation phase to write a new value. Since the tags are generated and propagated only by write operations then if a write changes the value of the system then it appends a tag already written, or the initial tag of the register.

Finally, to show that *evolution* is preserved, we observe that the version of a register is given by its tag, where tags are compared lexicographically (first the number $tag.ts$ and then the writer identifier to break ties). A successful write $\pi_1 = cvr\text{-}\omega(tag)[tag']$ generates a new tag tag' from tag such that $tag'.ts = tag.ts + 1$. Consider sequences of tags $tag_1, tag_2, \ldots, tag_k$ and $tag'_1, tag'_2, \ldots, tag'_\ell$ such that $tag_1 = tag'_1$. Assume that $cvr\text{-}\omega(tag_i)[tag_{i+1}]$, for $1 \leq i < k$, and $cvr\text{-}\omega(tag'_i)[tag'_{i+1}]$, for $1 \leq i < \ell$, are successful write operations. If $tag_1.ts = tag'_1.ts = z$, then $tag_k.ts = z + k$ and $tag'_\ell.ts = z + \ell$, and if $k < \ell$ then $tag_k < tag'_\ell$. □

Supporting Large Versioned Objects. Fan and Lynch [9], using algorithm MWABD as a building block, showed how large atomic R/W objects can be efficiently replicated. The main idea of their algorithm, called LDR, is to have two distinguished sets of servers: Replicas and Directories. Replica servers are the ones that actually store the object's data (value), while Directories keep track of the tags of the object and the associated Replicas that store the data of the object. A reader or writer first runs algorithm MWABD on the Directories to obtain the highest tag of the object, and the identity of the Replicas that have the associated value (aka, the most recent value of the object). A read operation, then contacts a subset of the Replicas to obtain the value of the object. A write sends the new value to a majority of the Replicas, while ensuring that Directories are updated (see [9] for details). By replacing algorithm MWABD with algorithm VMWABD and performing a few modifications to the Replicas, we can turn algorithm LDR into an algorithm that can handle *large versioned R/W objects*, such as large files. See Appendix C for the modified LDR.

7 Conclusion

In this paper we have introduced *versioned registers* and a new property for concurrent versioned registers, we call *coverability*. A versioned register associates a version with its value, and with each operation that wants to modify its value. An operation may modify the value and the version of the register, or it may just retrieve its value-version pair. Coverability defines the exact guarantees that a versioned register provides when it is accessed concurrently by multiple processes with respect to the evolution of its versions, over a total order of its operations. We introduce two levels of coverability: *strong* and *weak*. Strong coverability requires that only a single operation modifies each version of the register, whereas weak coverability is more relaxed allowing multiple concurrent operations to modify the same version.

We combine coverability with atomicity to obtain (strongly/weakly) coverable atomic registers. The successful writes on the register follow the total order of atomicity, while preserving the properties required by coverability. We note that a different total ordering could be used with coverability to obtain other types of "coverable objects". In fact, we believe it would be interesting to investigate further the use of coverable objects for the introduction of distributed algorithms for various applications. The fact that each operation is enhanced by the version of the object provides the flexibility to manipulate the effect of an operation under some conditions on the version of the object with respect to the version of the operation.

References

[1] AGUILERA, M. K., AND HORN, S. L. Abortable and query-abortable objects and their efficient implementation. In *In Proceedings of the 26th Symposium on Principles of Distributed Computing* (2007).

[2] ATTIYA, H., BAR-NOY, A., AND DOLEV, D. Sharing memory robustly in message passing systems. *Journal of the ACM 42(1)* (1996), 124–142.

[3] BOICHAT, R., DUTTA, P., FRØLUND, S., AND GUERRAOUI, R. Deconstructing paxos. *SIGACT News 34*, 1 (Mar. 2003), 47–67.

[4] CHOCKLER, G., DOBRE, D., AND SHRAER, A. Brief announcement: Consistency and complexity tradeoffs for highly-available multi-cloud store. In *The International Symposium on Distributed Computing (DISC)* (2013).

[5] CHOCKLER, G., AND MALKHI, D. Active disk paxos with infinitely many processes. *Distributed Computing 18*, 1 (2005), 73–84.

[6] DOBRE, D., VIOTTI, P., AND VUKOLIĆ, M. Hybris: Robust hybrid cloud storage. In *Proceedings of the ACM Symposium on Cloud Computing* (New York, NY, USA, 2014), SOCC '14, ACM, pp. 12:1–12:14.

[7] DUTTA, P., GUERRAOUI, R., LEVY, R. R., AND CHAKRABORTY, A. How fast can a distributed atomic read be? In *Proceedings of the 23rd ACM symposium on Principles of Distributed Computing (PODC)* (2004), pp. 236–245.

[8] ENGLERT, B., GEORGIOU, C., MUSIAL, P. M., NICOLAOU, N., AND SHVARTSMAN, A. A. On the efficiency of atomic multi-reader, multi-writer distributed memory. In *Proceedings 13th International Conference On Principle Of DIstributed Systems (OPODIS 09)* (2009), pp. 240–254.

[9] FAN, R., AND LYNCH, N. Efficient replication of large data objects. In *Distributed algorithms* (Oct 2003), F. E. Fich, Ed., vol. 2848/2003 of *Lecture Notes in Computer Science*, pp. 75–91.

[10] FISCHER, M. J., LYNCH, N. A., AND PATERSON, M. S. Impossibility of distributed consensus with one faulty process. *Journal of ACM 32*, 2 (1985), 374–382.

[11] GEORGIOU, C., NICOLAOU, N. C., AND SHVARTSMAN, A. A. On the robustness of (semi) fast quorum-based implementations of atomic shared memory. In *DISC '08: Proceedings of the 22nd international symposium on Distributed Computing* (Berlin, Heidelberg, 2008), Springer-Verlag, pp. 289–304.

[12] GEORGIOU, C., NICOLAOU, N. C., AND SHVARTSMAN, A. A. Fault-tolerant semifast implementations of atomic read/write registers. *Journal of Parallel and Distributed Computing 69*, 1 (2009), 62–79.

[13] HERLIHY, M. Wait-free synchronization. *ACM Trans. Program. Lang. Syst. 13*, 1 (1991), 124–149.

[14] HERLIHY, M. P., AND WING, J. M. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS) 12*, 3 (1990), 463–492.

[15] LAMPORT, L. How to make a multiprocessor computer that correctly executes multiprocess progranm. *IEEE Trans. Comput. 28*, 9 (1979), 690–691.

[16] LAMPORT, L. On interprocess communication, part I: Basic formalism. *Distributed Computing 1*, 2 (1986), 77–85.

[17] LYNCH, N. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[18] LYNCH, N., AND TUTTLE, M. An introduction to input/output automata. *CWI-Quarterly* (1989), 219–246.

[19] LYNCH, N. A., AND SHVARTSMAN, A. A. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proceedings of Symposium on Fault-Tolerant Computing* (1997), pp. 272–281.

[20] MAZIÈRES, D., AND SHASHA, D. Building secure file systems out of byzantine storage. In *Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing* (New York, NY, USA, 2002), PODC '02, ACM, pp. 108–117.

Appendix

A Impossibility of Implementing Weakly CoVerable Registers using Ranked Registers

In this section we provide the proofs to the lemmas presented in Section 4. Before proceeding to the proofs let us introduce some notation we use throughout this section.

Let R be a set of ranked registers. Let $R_x \subseteq R$ denote the set of ranked registers on which a process p_i performs a $rr\text{-}\omega(r)[*,*]_{p_i,*}$ during a coverable write operation π_x in an execution ξ. $R_x = cR_x \cup aR_x$, where cR_x is the set of ranked register such that p_i performs a $rr\text{-}\omega(r)[*, commit]_{p_i,*}$ that commits during π_x, and aR_x the set of ranked register such that p_i performs a $rr\text{-}\omega(r)[*, abort]_{p_i,*}$ that aborts during π_x. For any pair of write operations $\pi_x \rightarrow_\xi \pi_y$, let the set $R_{x,y} = R_x \cap R_y$ be the set of ranked registers such that both π_x and π_y perform a ranked write. We finally denote by $cR_{x,y} = cR_x \cap R_y$ and $aR_{x,y} = aR_x \cap R_y$, the set of registers where p_i committed (or aborted resp.) during π_x and they were also written during operation π_y.

Proof of Lemma 7. Let the weakly coverable register be implemented by k ranked registers each with a highest rank $r_1, r_2, \ldots, r_k$ respectively at the end of some execution fragment ξ. For the rest of the proof we will construct extensions of ξ. Also, let the state of the coverable object be (v, ver) at the end of ξ.

Assume to derive contradiction that we extend ξ with a write operation $\omega_1 = cvr\text{-}\omega(ver)[ver', chg]_{p_i}$ that revises the coverable register, and all the write operations performed during ω_1 on the ranked registers abort. From that it follows that for each write operation $rr\text{-}\omega(r)[r_j, abort]_{p_i,j}$ performed by p_i on some register j, $r_j > r$. Let the new execution be ξ'.

We extend ξ' with another write operation $\omega_2 = cvr\text{-}\omega(ver'')[ver''', chg]_{p_z}$ by process p_z to obtain execution ξ''. Since $\omega_1 \rightarrow_\xi \omega_2$ then by *consolidation*, $ver'' \geq ver'$, and $\omega_1 <_\xi \omega_2$. Moreover, since ω_1 is not concurrent with any other operation, then by consolidation ver' is the largest version introduced in ξ. Since, by *continuity*, ver'' has to be equal to a version introduced by a preceding operation, and since $ver'' \geq ver'$ (the largest version introduced), then ω_2 revises $ver'' = ver'$ to a newer version ver'''. Note however that for any write operation $rr\text{-}\omega(r')[r_j, *]_{p_z,j}$ performed on any of the ranked registers, for $1 \leq j \leq k$, the highest rank for j at the time of the write was r_j.

Finally consider the execution $\Delta\xi''$ that is similar to ξ'' without containing ω_1. In other words $\Delta\xi''$ extends ξ with the write operation ω_2. Observe that any write operation $rr\text{-}\omega(r')[r_j, *]_{p_z,j}$ performed by p_z on ranked register j during ω_2 observes a highest rank r_j as in ξ''. So if such a write committed (or aborted) in ξ'' will also commit (or abort) in $\Delta\xi''$ as well. Therefore, since ω_2 revised the value of the coverable register in ξ'' will revise the value of the coverable register in $\Delta\xi''$ as well. However, the last proceeding write operation is of the form $cvr\text{-}\omega(*)[ver, *]$ for $ver \neq ver'$. Thus $\Delta\xi''$ violates the *continuity* property and hence contradicts our initial assumption. □

Proof of Lemma 8. We will assume to derive contradiction that π_2 does not write on any ranked register that p_i wrote (and committed) during π_1. More formally, let cR_1 be the set of ranked registers s.t. for all $j \in cR_1$, $rr\text{-}\omega(r)[*, commit]_{p_i,j}$ for some rank r during π_1. Let R_2 be the set of ranked registers s.t. for all $q \in R_2$, p_z invokes $rr\text{-}\omega(r')[*,*]_{p_z,q}$ during π_2. Note that since π_1 revises the version of the object, then according to Lemma 7, $|cR_1| \geq 1$. According to our assumption $cR_1 \cap R_2 = \emptyset$.

Let us now construct an execution that contains the two operations π_1 and π_2. Consider an execution fragment ξ that ends with a state associated with a version ver. Let us assume that there exists an algorithm A that uses k ranked registers each with a highest rank $r_1, r_2, \ldots, r_k$ respectively at the end of ξ. We extend ξ with operation π_1 and obtain ξ_1. Since π_1 changes the version of the object, by Lemma 7, there exists a ranked register $j \in cR_1$ such that, p_i invokes an operation that commits on j, $rr\text{-}\omega(r)[*, commit]_{p_i,j}$, during π_1.

Next we extend ξ_1 by π_2 and obtain ξ_2. Since according to our assumption, $cR_1 \cap R_2 = \emptyset$, then it must be the case that the highest rank observed by π_2 in any $j \in R_2$ is r_j, i.e. the highest rank of j at the end of ξ. So it returns either r_j or r' the rank used by p_z. That includes also the ranked registers that p_i tried to modify and aborted during π_1.

Consider now the execution $\Delta\xi_2$ which is similar to ξ_2, without operation π_1. In particular, $\Delta\xi_2$ is obtained by extending ξ with π_2. Notice that since p_z does not communicate with p_i, then p_z appears in the same state in both ξ_2 and $\Delta\xi_2$ before invoking π_2. Thus, p_z attempts to write on the same set of ranked registers R_2 in both executions. Since ξ is extended by π_2 alone, then any write operation on the ranked registers $j \in R_2$ is r_j (as in ξ_2). So π_2 cannot distinguish ξ_2 from $\Delta\xi_2$ and thus revises ver_1 in $\Delta\xi_2$ as well. However, $\Delta\xi_2$ does not contain a $cvr\text{-}\omega(*)[ver_1, chg]$ operation, therefore π_2 violates the *continuity* property of weak coverability. This contradicts our assumption. □

Proof of Lemma 9. Consider again an execution fragment ξ that ends with a state associated with a version ver. Let us assume that there exists an algorithm A that uses a set $|R| = k$ of ranked registers each with a highest rank $r_1, r_2, \ldots, r_k$ respectively at the end of ξ. We know by Lemma 7, that each operation π_x that changes the version of the weakly coverable register performs a write that commits on at least a single ranked register in R.

We extend ξ with the following non-concurrent operations (listed in the order they take place) to obtain execution ξ_1:

- operation $\pi_1 = \pi(ver)[ver_1, chg]_{p_1}$
- operation $\pi_2 = \pi(ver_1)[ver_2, chg]_{p_2}$

By Lemma 8, $cR_1 \cap R_2 \neq \emptyset$. Assume to derive contradiction that $\forall j \in cR_1 \cap R_2$, p_1 performs a committed write with a rank $r_{\pi_1} > r_j$ and p_2 performs a write with a rank $r_{\pi_2} < r_{\pi_1}$ (that may commit or not). Since according to our assumption, $\forall j \in cR_1 \cap R_2$, the rank of p_2 has to be smaller than the rank used by p_1, we assume w.l.o.g. that p_2 uses the same rank r_2 for all the ranked writes.

By the order of operations in ξ_1 it follows that for all $j \in cR_1 \cap R_2$, $rr\text{-}\omega(r_{\pi_1})[r_j, commit]_{p_1,j}$ appears before $rr\text{-}\omega(r_{\pi_2})[r_1, *]_{p_2,j}$ in ξ_1. Moreover, observe that, by Definition 6, for each register $i \in R_1 - cR_1$, $r_i > r_{\pi_1}$ since the write from π_1 aborted. Since π_2 changes the version of the weakly coverable register, then by Lemma 7, $cR_2 \neq \emptyset$. Notice that, even though we assume that $r_2 < r_1$, the operations in ξ_1 may commit without violating the ranked register properties of Definition 6 (as a write operation with a smaller rank does not have to abort). In order to preserve weak coverability, π_2 changes the version ver_1 to ver_2.

Consider now the execution $\Delta\xi_1$ that contains the same operations but with π_1 and π_2 in reverse order. In particular $\Delta\xi_1$ extends ξ with operations:

- operation $\pi_2 = \pi(ver'_1)[ver'_2, chg]_{p_2}$
- operation $\pi_1 = \pi(ver)[ver_1, chg]_{p_1}$

Since there is no communication assumed between the processes then π_2 uses rank r_{π_2} in $\Delta\xi_1$ as well. It is easy to see that for any register $i \in R_2 - R_1$, π_2 observes the same highest rank r_i in both executions ξ_1 and $\Delta\xi_1$. So if the rank write of π_2 on those registers commits in ξ_1 then it also commits in $\Delta\xi_1$. So the only registers that may allow π_2 to differentiate between the two executions are the ones in the intersection $cR_1 \cap R_2$. There are two cases to consider: (i) $\forall j \in cR_1 \cap R_2$, $r_j > r_{\pi_2}$, and (ii) $\exists j \in cR_1 \cap R_2$, and $r_j \leq r_{\pi_2}$.

Case (i): In case (i), π_2 witnesses a higher rank from all the registers in $cR_1 \cap R_2$ as in ξ_1. So for each register $j \in cR_1 \cap R_2$, if $rr\text{-}\omega(r_{\pi_2})[r_j, *]_{p_2,j}$ committed in ξ_1 then the write commits in $\Delta\xi_1$ as well. Thus, π_2 will not be able to distinguish the two operations and it extends $ver'_1 = ver_1$ in $\Delta\xi_1$ as well. However, ver_1 is not written in $\Delta\xi_1$, thus π_2 violates *continuity* property and contradicts our assumption.

Case (ii): So it remains to examine the second case were $\exists j \in cR_1 \cap R_2$, and $r_j \leq r_{\pi_2}$. In this case $rr\text{-}\omega(r_{\pi_2})[r_j, *]_{p_2,j}$ has to commit in $\Delta\xi_1$. If the same operation committed in ξ_1 as well then π_2 cannot distinguish the two executions and thus violates coverability as shown before. Let us assume that π_2 did not commit in ξ_1. Hence, π_2 distinguishes $\Delta\xi_1$ from ξ_1. To preserve weak coverability, π_2 has to extend version $ver'_1 = ver$ to a version $ver'_2 > ver$. At the end of π_2 the highest rank of $r_j = r_{\pi_2}$. When π_1 is invoked it performs rank writes using rank r_{π_1}, since there is no communication between the processes. Since, according

to our assumption $r_{\pi_2} < r_{\pi_1}$, it follows that $rr\text{-}\omega(r_{\pi_1})[*,*]_{p_1,j}$ commits in both ξ_1 and $\Delta\xi_1$. Moreover, since for all the rest registers $i \in cR_1$, $r_i > r_{\pi_2}$, π_1 will witness the same highest rank r_i from each of those registers, in both executions. Thus, all the write operations on those registers π_1 will commit on all those registers, and thus, π_1 will not be able to distinguish $\Delta\xi_1$ from ξ_1. Since, however ,it extended ver in ξ_1, then it extends ver in $\Delta\xi_1$ as well. However, as $\pi_2 \rightarrow_\xi \pi_1$, then by consolidation, π needs to extend a version larger or equal to ver'_1. Since $ver < ver'_1$ then consolidation is violated. And this completes the proof. □

B Strong Coverability vs Consensus

Consensus [17] is defined as the problem where a set of fail-prone processes try to agree on a single value for an object. A consensus protocol must specify two operations: (i) $propose(v)_{p_i}$, used by the process p_i to propose a value v for the object, and (ii) $decide()_{p_i}$, used by the process p_i to decide the value of the object. Any implementation of consensus must satisfy the following three properties:

- **(1) CTermination:** Every correct process decides a value;
- **(2) CValidity:** Every correct process decides at most one value, and if it decides some value v, then v must have been proposed by some process;
- **(3) CAgreement:** All correct process must decide the same value.

We show that a strongly coverable atomic register is equivalent to a consensus object. To support this statement we first present an implementation of a consensus object using a strongly coverable register, and then we describe an implementation of a strongly coverable register assuming the existence of a consensus object. In the implementation of consensus that follows we assume that all the processes propose a value and they decide by the end of the propose operation. Thus we combine the two actions in one operation. Figure 5 presents the pseudocode of the implementation of a consensus object using a strongly coverable atomic register.

At each process $i \in \mathcal{I}$
Local Variables: $lcver \in Versions, lcval \in Values, flag \in \{chg, unchg\}$

function PROPOSE(v)
  $lcval \leftarrow v$
  $(lcval, lcver, flag) \leftarrow$ cvr-write$(lcval, ver_0)$
  return $lcval$

Figure 5: Consensus using Strongly Coverable Atomic Registers

We assume that ver_0 is the initial version of the coverable register. When each process begins executing the algorithm it issues a write operation trying to revise ver_0 and propose its own local value as the value to be decided. According to strong coverability only a single write operation cvr-write$(v, ver_0)(v, ver_1, chg)$ is going to succeed proposing its value, say v, and change the version of the register from ver_0 to some version ver_1. All the rest of the write operations will be of the form cvr-write$(v', ver_0)(v, ver_1, unchg)$ and thus will fail to change the value and version of the register. The write operation will return $(lcval, lcver, flag) = (v, ver_1, unchg)$ no matter what value they tried to propose, and each will be able to agree on value $lcval = v$ reaching this way agreement. This discussion yields the following theorem.

Theorem 14 *The construction in Figure 5 implements a consensus object.*

Figure 6 shows the implementation of a strongly coverable atomic register using a consensus object. For our implementation of consensus we assume that the consensus oracle runs a separate instance of consensus on each version of the object. Thus, the oracle accepts as inputs the version we want to revise as well as the $\langle v, ver' \rangle$ tuple that consists of the value we propose. When that value is not specified, the oracle returns the tuple decided on the instance associated with the given version. If no consensus was reached for a given version then the

oracle returns the tuple $\langle \perp, \perp \rangle$. To generate a new version a process calls the function generate-version(ver). This procedure produces a unique version larger than any previous version, each time is executed. A trivial implementation of this function is to append the given version with the unique id of the invoking process.

At each process $i \in \mathcal{I}$
Local Variables: $lcver, ver_{new} \in Versions$ initially ver_0; $lcval \in Values$; $P \in Values \times Versions$

function cvr-write(v, ver)
  $ver_{new} \leftarrow$ generate-version(ver)
  $P \leftarrow$ propose$(ver, \langle v, ver_{new} \rangle)$
  if $P.ver == ver_{new}$ **then**
    $lcver \leftarrow ver_{new}$
    return $\langle P, chg \rangle$
  else
    while $P.ver \neq \perp$ **do**
      $\langle lcval, lcver \rangle \leftarrow \langle P.val, P.ver \rangle$
      $P \leftarrow$ propose$(lcver, \perp)$
    end while
    return $\langle lcval, lcver, unchg \rangle$

function cvr-read()
  $P \leftarrow$ propose$(\perp, lcver)$
  while $P.ver \neq \perp$ **do**
    $\langle lcval, lcver \rangle \leftarrow \langle P.val, P.ver \rangle$
    $P \leftarrow$ propose$(\perp, lcver)$
  end while
  return $\langle lcval, lcver \rangle$

Figure 6: Strongly Coverable Atomic Registers using Consensus

Theorem 15 *The construction in Figure 6 implements a strongly coverable atomic register.*

Proof. We show that the algorithm satisfies two properties: (i) strong coverability and (ii) atomicity.

Strong coverability requires that only a single write operation changes each version of the register. Let us assume to derive contradiction that there exists a version ver of the object s.t. two operations $\pi_1 =$ cvr-write$(v, ver)(v, ver_1, chg)$ and $\pi_2 =$ cvr-write$(v', ver)(v, ver_2, chg)$ both revise ver leading to two potentially different versions ver_1 and ver_2. For this to be possible it means that $P.ver = ver_1$ for π_1 and $P.ver = ver_2$ for π_2. P however is the value decided by the consensus oracle. Since both π_1 and π_2 revise the same version ver then they both invoked the consensus oracle on the same instance of the version ver. Since the consensus oracle reaches agreement on a single value then it must be the case that P is the same for both π_1 and π_2, and hence $P.ver = ver_1 = ver_2$. This however contradicts our assumption. Thus, only a single write operation is able to modify each version and this preserves strong coverability.

Atomicity is trivially preserved by the write operations as they follow the total order imposed by the versions they change. Read operations are ordered in terms of the write operations since they invoke the consensus oracle until they reach the latest version of the object. A read operation ρ_1 does not return an older value than a preceding read ρ_2, since ρ_2 would reach an earlier or at most the same version as ρ_1 before completing. Thus, ρ_2 will return the same or an older value as desired. Finally, a write operation that does not change the version of the register it must be ordered with respect to the rest of the read operations. Such write also discovers the latest accepted version and thus, as before, it will return the same or a newer value than the one returned by a preceding read or unsuccessful write operation. □

C Supporting Large Files

Figure 7 depicts a modified version of the LDR algorithm [9], that implements versioned large objects.

---

tr-write$(val, ver = maxtag)$

get-metadata: Send query request to *directory servers* and wait for $(tag, location)$ responses from a majority of them. Select the $(tag, location)$ among the collected replies with the largest tag; let $\langle \tau, \mathcal{S} \rangle$ be this pair and the integer component of τ be z. Then:

If $\tau \neq ver$ then do the following:

put-metadata: Send $\langle \tau, \mathcal{S} \rangle$ to the *directory servers* and wait for a majority of them to reply. Once those replies are received set $\langle \tau_{new}, \mathcal{S}_{new} \rangle = \langle \tau, \mathcal{S} \rangle$.

get: Send *get object* request to $f + 1$ *replica servers* in $\mathcal{S}$ for the τ version of the object and wait for a single server to reply with x. Return $\langle x, \tau, unchg \rangle$.

If $\tau = ver$ then do the following:

put: Create a new tag $\tau_{new} = \langle z + 1, wid \rangle$ where wid is the unique identifier of the writer. Send $\langle \tau_{new}, val \rangle$ to $2f + 1$ *replica servers* and wait for $f + 1$ replies. Collect the identifiers of the servers that replied in a set $\mathcal{S}_{new}$.

put-metadata: Send $\langle \tau_{new}, \mathcal{S}_{new} \rangle$ to all the *directory servers* and wait for the majority of them to reply. Return $\langle val, \tau_{new}, chg \rangle$.

tr-read()

get-metadata: Send query request to *directory servers* and wait for $(tag, location)$ responses from a majority of them. Select the $(tag, location)$ among the collected replies with the largest tag; let $\langle \tau, \mathcal{S} \rangle$ be this pair and the integer component of τ be z.

put-metadata: Send $\langle \tau, \mathcal{S} \rangle$ to the *directory servers* and wait for a majority of them to reply

get: Send *get object* request to $f + 1$ *replica servers* in $\mathcal{S}$ for the τ version of the object and wait for a single server to reply with x. Return $\langle x, \tau \rangle$.

directory-server

On receipt of get-metadata message: Send the tag-locations pair $\langle \tau_s, \mathcal{S} \rangle$ stored locally.

On receipt of put-metadata message: Let $\langle \tau_m, \mathcal{S}_m \rangle$ be the tag-location pair enclosed in the received message and $\langle \tau_s, \mathcal{S} \rangle$ the local pair on the server. Compare the tags τ_m and τ_s. If $\tau_m > \tau_s$ and $|\mathcal{S}_m| \geq f + 1$ then store $\langle \tau_m, \mathcal{S}_m \rangle$ locally.

replica-server

On receipt of put message: Add the $\langle \tau_m, value \rangle$ pair enclosed in the message to the local set of available pairs and send an acknowledgement.

On receipt of get message: If the value associated with the requested tag is in the set of pairs stored locally, respond with the value. Otherwise ignore the message.

Figure 7: Operations of the modified LDR algorithm