CoVer-ability: **Co**nsistent **Ver**sioning in Asynchronous, Fail-Prone, Message-Passing Environments

Nicolas Nicolaou*, Antonio Fernández Anta*, Chryssis Georgiou†
* 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

***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 the *version* (and thus value) 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. To preserve the order of versions, traditional approaches use locking, compare-and-swap (CAS), or linked-load/conditional-store (LL/SC) primitives to allow a single modification at a time on such objects. Such primitives however can be used to solve consensus, and thus are impossible to be implemented in an asynchronous, message-passing environment with failures.**

***Coverability*, relaxes the strong requirements imposed by stronger primitives, and allows us to define and implement consistent versioning in the aforementioned adversarial environment. In particular, coverability allows *multiple operations* to modify the same version of an object concurrently, leading to a set of different versions. Given an order of operations, coverability properties specify a single version in that set that any subsequent operation may modify, preserving this way the consistent evolution of the object. We first define *versioned* objects and then provide the specification of *coverability*. We then combine coverability with atomic guarantees to yield *coverable atomic read/write registers*; we show that coverable registers cannot be implemented by similar types of registers, such as ranked-registers. Next, we show how coverable registers may be implemented by modifying an existing MWMR atomic register implementation, and we continue by showing that coverable registers may be used to implement basic (weak) read-modify-write and file objects.**

I. 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 [2], [6], [15]–[17] 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 [16], [17] 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 [15], an extension of atomicity to general concurrent objects. More recent developments have proposed abortable operations in the event of concurrency [2], and ranked registers [6] that allow operations to abort in case a higher "ranked" operation was previously or concurrently executed in the system.

Although consistency semantics strictly specify the "placement" of events on an execution trace based on their timing characteristics, in many cases they are oblivious of the state of the object at the point when an event takes effect. For example, a write operation ω on a read/write register is ordered after all the writes that completed before ω, irrespective to the value that ω writes on the register. 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. Such guarantees are easy to achieve in systems that readily provide atomic compare-and-swap (CAS), or linked-load/conditional-store (LL/SC) operations. Such primitives allow modify operations to atomically obtain the current version and value of an object, modify both, and store the new version along with the new value of the object. As shown by Herlihy in [14], CAS can be used to solve consensus as it has a consensus number infinite. However, as shown by Fischer, Lynch and Paterson [11], solving consensus in an asynchronous, message-passing, fail-prone environment is impossible in the existence of a single crash failure. So the main question we will try to address is: *Can we provide versioning guarantees in an asynchronous, message-passing, fail-prone environment using weaker primitives, like read/write registers?*

A seminal work by Attiya, Bar Noy and Dolev [3], demonstrated that it is possible to introduce atomic read/write

This work is 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.

978-1-5090-3216-7/16/$31.00

registers in an asynchronous, message-passing environment where processes may fail. As noted before, in existing atomic read/write register implementations, write operations are 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 [3], [8], [12], [13], 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 [9], [20]. 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 [4] and ranked [6] 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 III).

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. Coverability 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 [21]. However, in contrast to [21], coverability allows processes that change the same version of the object to see the changes of each other in subsequent operations. In particular, by 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. The rest of the branches are discarded. This resembles the way that the forks in a bitcoin blockchain converge [1]. In particular, forks in a blockchain are created when two miners generate a new block concurrently. Both blocks are legitimate and each miner results in a different branch, rooted from the same blockchain. Miners tend to quickly converge on one chain and discard the other because of profit-related motives. These discarded chains are usually only one block long and are considered a statistical loss. In contrast to the "profit-related" motives of the bitcoin, coverability specifies which of the branches need to be discarded based on a provable ordering of the events. Notice that the stronger form of coverability where modify operations are totally ordered, avoids branching of the versions. However such primitive is equivalent with strong primitives like CAS and LL/SC, and thus it is as powerful as consensus (details can be found in [22]). Hence, it is challenging to implement strong coverability in some distributed systems, and impossible in an asynchronous system prone to failures (from the FLP result [11]).

An interesting property of coverability is that it is defined over a given order of events. Therefore coverability can be defined over the ordering yielded by any consistency scheme. In this paper we combine coverability with atomic guarantees and we obtain *coverable atomic read/write registers*. Coverable atomic registers have very interesting features. At first, they provide strong atomic guarantees, i.e they surpass weaker consistency guarantees like regularity [16], or eventual consistency [23], and in addition provide guarantees on the evolution of the value of the register. This allows coverable atomic registers to be used for the implementation of more complex objects like: (i) interesting *weak read-modify-write (RMW) objects* which in turn can be used to implement (ii) *file objects* (Section VI). Furthermore, we show they cannot be implemented using similar register types such as ranked registers (Section IV). And last but not least, they can be implemented in message passing asynchronous distributed systems where processes can fail, with a simple modification of existing atomic read/writer register implementations (Section V).

II. Model

We consider a distributed system composed of n *asynchronous* processes, with identifiers from a set $\mathcal{I} = \{p_1, \ldots, p_n\}$, that communicate by exchanging messages. A subset of processes in $\mathcal{I}$ may fail by *crashing*.

Processes can be modeled in terms of I/O Automata [19]. 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 of A. 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*.

III. Coverable Atomic Read/Write Registers

In this section we define a new type of read/write (R/W) register, the *versioned register*. Next we provide a new consistency property for concurrent versioned registers called *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 R/W 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 $\sigma' = (val', ver') \in V \times Versions$ 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, 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 *coverability* properties over a valid execution ξ of versioned registers with respect to some total order $>_\xi$ on the operations of ξ.

Definition 2 (Coverability): A valid execution ξ is **coverable** with respect to a total order $<_\xi$ on operations in $\mathcal{W}_{\xi,succ}$ if:

- (**Consolidation**) If $\pi_1 = cvr\text{-}\omega(*)[ver_i], \pi_2 = cvr\text{-}\omega(ver_j)[*] \in \mathcal{W}_{\xi,succ}$, 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 $\pi_2 = cvr\text{-}\omega(ver)[ver_i] \in \mathcal{W}_{\xi,succ}$, then there exists $\pi_1 \in \mathcal{W}_{\xi,succ}$ s.t. $\pi_1 = cvr\text{-}\omega(*)[ver]$ 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 $cvr\text{-}\omega(ver'_i)[ver'_{i+1}] \in \mathcal{W}_{\xi,succ}$, for $1 \leq i < k$, and $cvr\text{-}\omega(ver''_i)[ver''_{i+1}] \in \mathcal{W}_{\xi,succ}$, for $1 \leq i < \ell$, and $k < \ell$, then $ver' < ver''$.

Intuitively, *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. *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.

By Definition 2, 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 2 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 $\pi_1 = cvr\text{-}\omega(*)[ver_i] \in \mathcal{W}_{\xi,succ}$, 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 $cvr\text{-}\omega(*)[ver_i] <_\xi cvr\text{-}\omega(*)[ver_j]$ (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 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. 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 [15], or equivalently *atomic* (as defined specifically for registers by [17], [18]) if the following conditions are satisfied by any execution ξ of an implementation of the object.

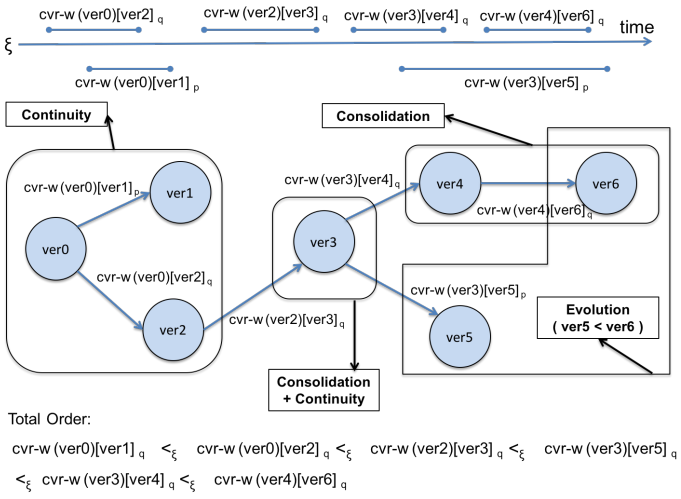

Fig. 1: Tree Illustration from Coverable Execution

Definition 3 (Atomicity): [18, 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 3, 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 4 (Coverable atomic register): A versioned register is **coverable** and **atomic**, referred as *coverable atomic register*, if any execution ξ on the register satisfies: (i) atomicity (Definition 3), and (ii) coverability (Definition 2) 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*.

IV. Coverable Atomic Registers vs Ranked Registers.

A type of registers that at first might resemble coverable registers are *ranked-registers* [6]. As we show here, ranked-registers are weaker than coverable registers. In particular, we

show that it is impossible to implement coverable registers using ranked-registers; we begin by providing a formal definition of ranked-registers.

Definition 5 (Ranked-Registers [6]): 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 coverable register. As in Section II, 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 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 [22].*

Lemma 6: Suppose there exists an algorithm A that implements a 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 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 7: 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 6), and two non-concurrent coverable write operations need to invoke a ranked write operation on a common rank register (Lemma 7). Using now Lemma 7 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 8: 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 coverable register cannot be implemented with ranked registers as those were defined in [6].

Theorem 9: There is no algorithm that implements a coverable register using a set of ranked registers.

Proof: The theorem follows from Lemmas 6, 7, and 8, 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 6 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 8 may lead to violation of the consolidation and continuity properties, and thus violation of coverability. ■

Observe that the key fact that makes ranked registers weaker than 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 [6] *non-fault-tolerant* ranked registers may preserve the total order of the ranks, and thus be used to implement consensus. As we show in [22] such ranked registers (i.e., that implement consensus) could be used to implement strongly coverable registers.

V. Implementing Coverable Atomic Read/Write Registers

We now show how we can implement coverable atomic registers. We do so by enhancing the Multi-Writer version of algorithm ABD [3], [20] (referred as MWABD) to preserve the properties of 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 could also adopt implementations of stronger objects like the ones presented in [4]–[7] 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 (replicas) $\mathcal{S} \subset \mathcal{I}$ and it uses $\langle tag, value \rangle$ pairs to order the *read* and *write* operations. A tag consists of a *non-negative integer* 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 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 2. 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 2. Note that the version parameter of the write operation is equal to the maximum tag that the invoking process witnessed.

Theorem 10: Algorithm vMWABD implements 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 to a majority of replicas before completing. It remains to show that vMWABD also satisfies validity and coverability.

Validity is satisfied since each tag is unique, as it is composed by an integer ts and the id of a process wid. 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 w_i discovers the maximum tag $\langle ts, w_j \rangle$ among the replicas and in the second phase it generates a tag $\langle ts + 1, w_i \rangle$. 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

```
1: at each writer w_i
2: Components:
3: maxP ∈ ℕ⁺ × 𝒲 × V, tg ∈ ℕ⁺ × 𝒲, v ∈ V, flag ∈ {chg, unchg}
4: Initialization:
5: tg ← ⟨0, w_i⟩, v ← ⊥, maxP ← ⟨tg, v⟩
6: function CVR-WRITE(val, ver)
7:     send (Query) to all servers                          ▷ Query Phase
8:     wait until (|𝒮|+1)/2 servers reply
9:     maxP ← max({m.⟨tg', v'⟩ | m received from some server})
10:    if ver = maxP.tg' then
11:        tg ← ⟨maxP.tg'.ts + 1, w_i⟩; v ← val; flag ← chg
12:        send (Write, ⟨tg, v⟩) to all servers              ▷ Write Phase
13:        wait until (|𝒮|+1)/2 servers reply
14:    else
15:        tg ← maxP.tg'; v ← maxP.v'; flag ← unchg
16:        send (Propagate, maxP) to all servers             ▷ Propagate Phase
17:        wait until (|𝒮|+1)/2 servers reply
18:    end if
19:    return(⟨tg, v⟩, flag)
20: end function

21: at each reader r_i
22: Components:
23: maxP ∈ ℕ⁺ × 𝒲 × V
24: function CVR-READ( )
25:    send (Query) to all servers                          ▷ Query Phase
26:    wait until (|𝒮|+1)/2 servers reply
27:    maxP ← max({m.⟨tg', v'⟩ | m received from some server})
28:    send (Propagate, maxP) to all servers                ▷ Propagate Phase
29:    wait until (|𝒮|+1)/2 servers reply
30:    return(maxP)
31: end function

32: at each server s_i
33: Components:
34: tg ∈ ℕ⁺ × 𝒲, v ∈ V
35: Initialization:
36: tg ← ⟨0, ⊥⟩, v ∈ V
37: function RCV(M)_q                                       ▷ Reception of a message from q
38:    if M.type ≠ Query and M.tg > tg then
39:        ⟨tg, v⟩ ← ⟨M.tg, M.v⟩
40:    end if
41:    send(⟨tg, v⟩) to q
42: end function
```

Fig. 2: The operations of algorithm vMWABD.

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 maximum tag $tag_{max} \geq tag_1$. Since ω_2 also changes the value and version of the register it means that its local tag tag_2 is equal to tag_{max}. This shows immediately that $tag_2 \geq tag_1$.

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.

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 writes. 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 [10], 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 [10] 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 [22] for the modified LDR.

VI. Applications of 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. 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 3 presents an implementation of a weak RMW object using coverable atomic R/W registers. We assume that the object offers a 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.

```
1: At each process i ∈ I
2: State Variables:
3: lcver ∈ Versions; oldval, lcval, newv ∈ Values;
4: flag ∈ {chg, unchg}
5: function RMW(F)
6:     ⟨oldval, lcver⟩ ← cvr-read()
7:     newv ← F(oldval)
8:     ⟨lcval, lcver, flag⟩ ← cvr-write(lcver, newv)
9:     if flag = chg then
10:        return ⟨lcval, success⟩
11:    else
12:        return ⟨lcval, fail⟩
13:    end if
14: end function
```

Fig. 3: Weak RMW using Coverable Atomic Registers

Theorem 11: The construction in Figure 3 implements a weak RMW object.

Proof: Consider an execution ξ of the algorithm. We begin the proof by studying the case where an operation 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 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 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 [14], they are impossible to be implemented in an asynchronous system with a single crash

```
1: At each process i ∈ 𝓘
2: State Variables:
3: lcver ∈ Versions
4: lcval ∈ Values
5: flag ∈ {chg, unchg}
6: Initialization:
7: lcver ← ver_0; lcval ← ⊥
8: function GET( )
9:     ⟨lcval, lcver⟩ ← cvr-read()
10:    return ⟨lcval, lcver⟩
11: end function
12: function REVISE(v, ver)
13:    ⟨lcval, lcver, flag⟩ ←
14:          cvr-write(ver, v)
15:    if flag = chg then
16:       return OK
17:    end if
18:    return ⟨lcval, lcver⟩
19: end function
```

Fig. 4: File Object using Coverable Atomic Registers

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 4 presents 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 4 we may conclude the following theorem.

Theorem 12: The construction in Figure 4 implements a file object.

VII. 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 combine coverability with atomicity to obtain 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] What is bitcoin fork? http://blog.cex.io/bitcoin-dictionary/what-is-bitcoin-fork-14622. Accessed: 2016-05-05.

[2] Aguilera, M. K., and Horn, S. L. Abortable and query-abortable objects and their efficient implementation. In *Proc. of PODC 2007,* pp. 23–32.

[3] Attiya, H., Bar-Noy, A., and Dolev, D. Sharing memory robustly in message passing systems. *Journal of the ACM 42(1)* (1996), 124–142.

[4] Boichat, R., Dutta, P., Frølund, S., Guerraoui, R. Deconstructing paxos. *SIGACT News 34*, 1 (2003), 47–67.

[5] Chockler, G., Dobre, D., and Shraer, A. Brief announcement: Consistency and complexity tradeoffs for highly-available multi-cloud store. In *Proc. of DISC 2013*.

[6] Chockler, G., and Malkhi, D. Active disk paxos with infinitely many processes. *Distributed Computing 18*, 1 (2005), 73–84.

[7] Dobre, D., Viotti, P., and Vukolić, M. Hybris: Robust hybrid cloud storage. In *Proc. of SOCC 2014*.

[8] Dutta, P., Guerraoui, R., Levy, R. R., and Chakraborty, A. How fast can a distributed atomic read be? In *Proc. of PODC 2004,*

[9] 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 *Proc. of OPODIS 2009,* pp. 240–254.

[10] Fan, R., and Lynch, N. Efficient replication of large data objects. In *Proc. of DISC 2003,* pp. 75–91.

[11] 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.

[12] Georgiou, C., Nicolaou, N. C., and Shvartsman, A. A. On the robustness of (semi) fast quorum-based implementations of atomic shared memory. In *Proc. of DISC 2008,* pp. 289–304.

[13] 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.

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

[15] Herlihy, M. P., and Wing, J. M. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst. 12,* 3 (1990), 463–492.

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

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

[18] Lynch, N. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[19] Lynch, N., and Tuttle, M. An introduction to input/output automata. *CWI-Quarterly* (1989), 219–246.

[20] Lynch, N. A., and Shvartsman, A. A. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proc. of FTC 1997,* pp. 272–281.

[21] Mazières, D., and Shasha, D. Building secure file systems out of byzantine storage. In *Proc. of PODC 2002,* pp. 108–117.

[22] Nicolaou, N., Fernández Anta, A., and Georgiou, C. Cover-ability: Consistent versioning for concurrent objects. *CoRR abs/1601.07352* (2016).

[23] Vogels, W. Eventually consistent. *Commun. ACM 52*, 1 (Jan. 2009), 40–44.