

# Dynamic Reconfiguration: Abstraction and Optimal Asynchronous Solution\*

Alexander Spiegelman<sup>1</sup>, Idit Keidar<sup>1</sup> and Dahlia Malkhi<sup>2</sup>

<sup>1</sup> Viterbi Dept. of Electrical Engineering, Technion, Haifa, Israel.

<sup>2</sup> VMware Research, Palo Alto, USA.

---

## Abstract

Providing clean and efficient foundations and tools for reconfiguration is a crucial enabler for distributed system management today. This work takes a step towards developing such foundations. It considers classic fault-tolerant atomic objects emulated on top of a static set of fault-prone servers, and turns them into dynamic ones. The specification of a dynamic object extends the corresponding static (non-dynamic) one with an API for changing the underlying set of fault-prone servers. Thus, in a dynamic model, an object can start in some configuration and continue in a different one. Its liveness is preserved through the reconfigurations it undergoes, tolerating a versatile set of faults as it shifts from one configuration to another.

In this paper we present a general abstraction for asynchronous reconfiguration, and exemplify its usefulness for building two dynamic objects: a read/write register and a max-register. We first define a dynamic model with a clean failure condition that allows an administrator to reconfigure the system and switch off a server once the reconfiguration operation removing it completes. We then define the Reconfiguration abstraction and show how it can be used to build dynamic registers and max-registers. Finally, we give an optimal asynchronous algorithm implementing the Reconfiguration abstraction, which in turn leads to the first asynchronous (consensus-free) dynamic register emulation with optimal complexity. More concretely, faced with  $n$  requests for configuration changes, the number of configurations that the dynamic register is implemented over is  $n$ ; and the complexity of each client operation is  $O(n)$ .

**Regular submission.** If not selected, please consider for the brief announcement format.

**Eligible to be considered for the best student paper award:** Alexander Spiegelman, the primary contributor, is a full-time student.

**1998 ACM Subject Classification** F.1.2 Modes of Computation

**Keywords and phrases** Reconfiguration, Dynamic Objects, Optimal Algorithm

**Digital Object Identifier** 10.4230/LIPIcs.CVIT.2016.23

## 1 Introduction

The goal of this paper is to take a static fault-tolerant object like an atomic read/write register and turn it into a dynamic fault-tolerant one. A static object exposes an API (e.g., read/write) to its clients, and is emulated on top of a set of fault-prone servers (sometimes called base objects) via protocols like ABD [5]. We refer to the underlying set of fault-prone servers as a *configuration*. To convert a static object into a dynamic one, we first extend the

---

\* Alexander Spiegelman is grateful to the Azrieli Foundation for the award of an Azrieli Fellowship.



object's API to support *reconfiguration*. Such an API is essential for administrators, who should be able to remove old or faulty servers and add new ones without shutting down the service. One of the challenges in formalizing dynamic models is to define a precise fault condition, so that an administrator who requests to remove a server  $s$  via a reconfiguration operation will know when she can switch  $s$  off without risking losing the object's state (e.g., the last written value to a read/write register).

To this end, we first define a clean dynamic failure model, in which an administrator can immediately switch a server  $s$  off once a reconfiguration operation that removes  $s$  completes. Then, we provide an abstraction for consensus-less reconfiguration in this model. To demonstrate the power of our *Reconfiguration* abstraction we use it to implement two dynamic atomic objects. First, we focus on the basic building block of a read/write register; thus, other (static) objects that can be emulated from read/write registers (e.g., atomic snapshots) can be made dynamic by replacing the underlying registers with dynamic ones. Second, we emulate a max-register [4], which on the one hand can be implemented asynchronously [5, 12] (on top of fault-prone servers), and on the other hand cannot be emulated (for an unbounded number of clients) on top of a bounded set of read/write registers<sup>1</sup> [12, 4]. Thus, a standalone implementation of dynamic max-registers is required.

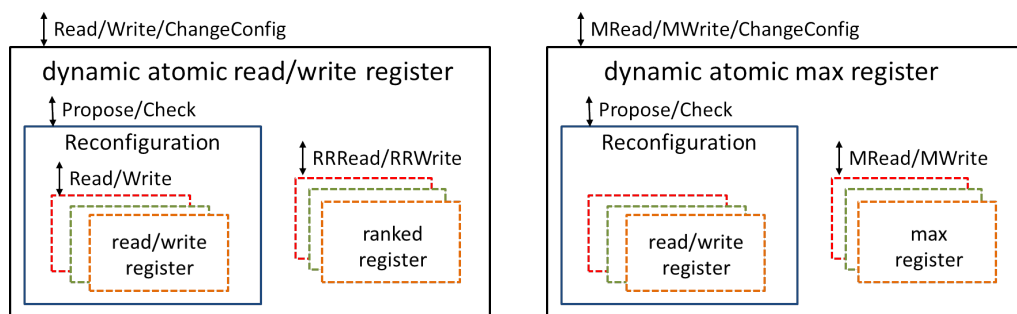
**Complexity.** We present an optimal-complexity implementation of our Reconfiguration abstractions in asynchronous environments, which in turn leads to the first optimal implementation of a dynamic read/write register in this model. More concretely, faced with  $n$  administrator reconfiguration requests, the number of configurations that the dynamic object is implemented over is  $n$ ; and the number of rounds (when the algorithm accesses underlying servers) per client operation is  $O(n)$ . A comparison with previous solutions appears in Section 2.

**Dynamic fault model.** In Section 3 we provide a succinct failure condition capturing a versatile set of faults under which the dynamic object's liveness is guaranteed. We define the dynamic fault model as an interplay between the object's implementation and its environment: New configurations are *introduced* by clients, (which are part of the object's environment). The object implementation then *activates* the requested configuration, at which point old configurations are *expired*. Between the time when a configuration is introduced and until it is expired, the environment can crash at most a minority of its servers. For example, when reconfiguring a register from configuration  $\{A, B, C\}$  into  $\{D, E, F\}$ , initially a majority of  $\{A, B, C\}$  must be available to allow read/write operations to complete. Then, when reconfiguration is triggered,  $\{D, E, F\}$  is introduced, and subsequently, majorities of both configurations must be available, to allow state-transfer to occur. Finally, when the reconfiguration operation completes, leading to  $\{D, E, F\}$ 's activation,  $\{A, B, C\}$  is expired, and every server in it may be immediately shutdown.

**Reconfiguration abstraction.** Since a configuration is a finite set of servers, we can use ABD [5] to emulate in each configuration a set of (static) atomic read/write registers (as well max-registers), which are available as long as the configuration is not expired. The Reconfiguration abstraction, in contrast, is not tied to a specific configuration, but rather abstracts away the coordination among clients that wish to change the underlying set of servers (configuration) emulating the dynamic object. Its specification, which is formally

---

<sup>1</sup> A max-register for  $k$  clients requires at least  $k$  read/write registers [12].



(a) Dynamic atomic read/write register on top of the Reconfiguration abstraction. (b) Dynamic atomic max-register on top of the Reconfiguration abstraction.

■ **Figure 1** The Reconfiguration abstraction usage. Solid (dashed) blocks depict dynamic (resp. static) objects.

defined in Section 4, exposes two API methods, *Propose* and *Check*. Clients use *Propose* to request changes to the configuration, and *Check* to learn of changes proposed by other clients. Both return a configuration and a set of *speculations*. The returned configuration reflects all previous proposals and possibly some ongoing ones. The less obvious return value of *Reconfiguration* is the speculation set. This set is required since there is no guarantee that all clients see the same sequence of configurations (indeed, *Reconfiguration* is weaker than consensus). Therefore, a dynamic object implementation that uses *Reconfiguration* needs to read from every configuration that *Check* returned to any *other* client, and transfer the most up-to-date value read in any of these to the new configuration returned from *Check*. To this end, *Reconfiguration* returns a speculation set that includes all configurations previously returned to all clients (and possibly additional proposed ones).

In Section 5, we implement (1) a dynamic atomic read/write register on top of the *Reconfiguration* abstraction and static atomic ranked registers [11] (one in every configuration), and (2) a dynamic atomic max-register on top of *Reconfiguration* and static atomic max-registers. See figure 1 for illustrations. In Section 6 we give an optimal consensus-less algorithm for *Reconfiguration*, which together with the read/write register emulation of Section 5 yields an optimal dynamic read/write register algorithm.

In summary, this paper makes three contributions: it defines a failure condition that allows an administrator to shutdown removed servers; it introduces the *Reconfiguration* abstraction, which captures the essence of reconfiguration; and it presents an asynchronous optimal-complexity solution for dynamic atomic registers. Section 7 concludes the paper, and formal correctness proofs of all algorithms are given in Appendix A.

## 2 Related Work

**Model.** The problem of object reconfiguration has gained growing attention in recent years [15, 20, 3, 21, 18, 14, 24, 13, 22, 17, 23, 6, 7]. However, dynamic failure models do not always make it clear when exactly an administrator can shutdown a removed server. Early works supporting dynamic objects [20, 15, 10] simply assume that a configuration is available as long as some client may try to access it. *SmartMerge* [18] uses a shared non-reconfigurable auxiliary object (lattice agreement) that is forever available to all clients, meaning that a majority of the servers emulating this auxiliary object can never be switched off. *DynaStore* [3] was the only previous work to define dynamic failure conditions based

on a reconfiguration API, but these conditions are complicated, and restrict reconfiguration attempts as well as failures. Moreover, DynaStore does not separate clients from servers as we do here. Following [13, 18], we formulate the problem in shared memory, which makes it easier to reason about and clearer.

Other works [6, 7] assume a broadcast mechanism for announcing joins instead of an API for adding and removing processes, and bound the rate of changes of the underlying set of servers; the latter is necessary if one wants to ensure liveness for all operations (as [6] does) – no asynchronous reconfigurable service can ensure liveness unless the reconfiguration rate is limited in some way [22]. Like many earlier works [3, 13, 18], we do not explicitly bound the reconfiguration rate, and hence ensure liveness only if the number of reconfigurations is finite.

**Abstractions.** All previous works have considered reconfiguration in some specific context – state machine replication [19, 8, 9], read/write register emulation [3, 18, 13, 15], or atomic snapshot [23]. To the best of our knowledge, this work is the first to specify general dynamic objects as extensions of their static counterparts and to provide a general abstraction for dynamic reconfiguration. We note that although [13] define a reconfigure primitive intended to capture the core reconfiguration problem, that primitive is not sufficiently strong for implementing an atomic register, (in particular, since it does not require real-time order), and indeed, they do not implement their atomic register on top of it.

**Dynamic register complexity.** In a recent non-refereed tutorial [24], we give a generic formulation that allows us to compare the complexity of different algorithms [15, 20, 18, 13, 3], as follows: Given that  $n$  is the number of proposed configuration changes and  $m$  is the total number of operations (read/write/reconfig) invoked on the atomic register, DynaStore [3] goes through  $O(\min(mn, 2^n))$  configurations, and requires a constant number of operations in every configuration, so  $O(\min(mn, 2^n))$  is also DynaStore’s operation complexity. Parsimonious SpSn [13] reduces the number of traversed configurations to  $O(n)$ , but since they invoke a linear number of operations in every configuration, their total operation complexity is  $O(n^2)$ .

Now notice that it is always possible to stagger reconfiguration proposals in a way that forces the system to go through  $\Omega(n)$  configurations. The asymptotically optimal  $O(n)$  operation complexity is straightforward to achieve in consensus-based solutions [15, 20, 10]. This complexity was also achieved by SmartMerge [18], but this was done using an auxiliary object that was assumed to be live indefinitely, i.e., was not reconfigurable in itself. Our algorithm is the first consensus-free and fully reconfigurable dynamic register algorithm with optimal complexity.

### 3 Dynamic Model

We consider a fault-prone shared memory model [16]: The system consists of an infinite set  $\Pi$  of *clients* (sometimes called processes), any number of which may fail by crashing, and an infinite set  $\Phi$  of *servers* (sometimes called base objects) supporting arbitrary atomic low-level objects. Clients access servers via low level operations (e.g., read/write), which may take arbitrarily long to arrive and complete, hence the system is asynchronous.

We address in the paper two atomic objects: a classical fault tolerant read/write register and a max-register [4]. Both registers provide clients with two API methods: Read and Write in case of read/write register, and MRead and MWrite in case of max-register. In a well-formed execution, a client invokes API methods one at a time, though calls by different

clients may be interleaved in real time. For a well-formed execution, there exists a serialization of all client operations that preserves the operations' real time order, such that (1) in case of read/write register a Read returns the value written in the latest Write preceding it, or  $\perp$  if there is no preceding Write; and (2) in case of max-register an MRead returns the highest value written by an MWrite that precedes it, or  $\perp$  if there is no preceding MWrite. (In case of max-registers, the values domain is ordered.)

**Configurations.** The universe of servers is infinite, but at any moment in time, a client chooses to interact with a subset of it. In our model, a *configuration* is a set of included and excluded servers, where configuration *membership* is the set of included and not excluded servers in the configuration. Formally:

<i>Changes</i>	$\triangleq$	$\{+s \mid s \in \Phi\} \cup \{-s \mid s \in \Phi\}$
<i>Configuration</i>	$\triangleq$	subset of <i>Changes</i>
<i>C.membership</i>	$\triangleq$	$\{s \mid +s \in C \wedge -s \notin C\}$

For example  $C = \{+s_1, +s_2, -s_2, +s_3\}$  is a configuration representing the inclusion of servers  $s_1, s_2$ , and  $s_3$ , and the exclusion of  $s_2$ , and  $C.membership$  is  $\{s_1, s_3\}$ . Tracking excluded servers in addition to the configuration's membership is important in order to reconcile configurations suggested by different clients. The configuration size is the number of changes it includes— in this example,  $|C| = 4$ .

**Dynamic fault model.** A dynamic fault model is an interplay between the adversary's power and the following events, which are invoked as part of client operations:

**introduce( $C$ ):** indicates that  $C$  is going into use; and

**activate( $C$ ):** indicates that the state transfer to  $C$  is complete.

By convention we say that the initial configuration  $C_{init}$  is introduced and activated at time 0.

The above events govern the life-cycle of configurations. A configuration  $C$  becomes *activated* once an activate( $C$ ) event occurs. Note that not all introduced configurations are necessarily activated at some point. A configuration  $C$  becomes *expired* once activate( $D$ ) occurs s.t.  $C$  does not contain  $D$ . Intuitively,  $D$  reflects events (inclusions or exclusions) that are not reflected in  $C$ , and hence  $C$  has become “outdated”. Our algorithm will enforce a containment order among activated configurations, and will thus ensure that the latest activated one is not expired.

The following two conditions constrain the power of the adversary:

► **Definition 1.** (liveness conditions)

**Availability:** The adversary can crash at most a minority of  $C.membership$  between the time when introduce( $C$ ) occurs and until  $C$  is expired.

**Weak Oracle:** When a client interacts with an expired configuration  $C$ , it either receives responses to calls from a majority of  $C.membership$ , or returns an exception notification  $\langle error, D \rangle$  for some activated  $D$ , where  $C \not\supseteq D$ .

Note that such an oracle (sometimes called directory service) is inherently required in order to allow slow clients to find non-expired configurations in an asynchronous system where old configurations may become unavailable [2, 23]. Our oracle definition is weak— in particular, the activated configuration it returns may itself be expired, and different clients

may get different responses; it can be trivially implemented using a broadcast mechanism as assumed in some previous works [6, 7], and trivially holds if configurations must remain available as long as some client may access them, as in other previous works [15, 20, 10].

**Static versus dynamic objects.** A *static object* is one in which clients interact with a fixed configuration. In order to disambiguate a static object, scoped within a configuration  $C$ , from a dynamic one, we will label the methods of a static object with a “ $C$ .” For every configuration  $C$ , as long as a majority of  $C.membership$  is alive, clients can use ABD [5] to emulate (static) atomic registers on top of the servers in  $C.membership$ . We denote:

$C.x \leftarrow value$	A Write( $value$ ) operation to register $x$ in configuration $C$
$C.x$	A Read of $x$
$C.collect(array)$	A bulk Read of all the registers in $array$

Since a complete array can be collected from servers using ABD in the same number of rounds as reading a single variable, we count a collect as a single operation for complexity purposes. Note that each register in the array is atomic in itself, but the collect is not atomic.

The methods of a dynamic object are not scoped with any configuration; it can start in some configuration and continue in a different one. A dynamic object’s API includes a `ChangeConfig` operation that allows clients to change the set of servers implementing the object. The implementation of `ChangeConfig` is object-specific, because it needs to transfer the state of the object across configurations, e.g., the last written value in case of an atomic register.

Clients pass to `ChangeConfig` a parameter  $Proposal \subset Changes$  containing a proposed set of configuration changes. `ChangeConfig` returns a configuration  $C$  s.t. (1)  $C$  is activated, (2)  $C \supseteq Proposal$ , and (3) every configuration introduced or activated by `ChangeConfig` consists of  $C_{init}$  plus a subset of changes proposed by clients before the operation returns.

The liveness guarantee of a dynamic object is that, assuming the number of `ChangeConfig` proposals is finite, every correct client’s operation eventually completes. Note that if the number of `ChangeConfig` proposals is infinite, it is impossible to ensure liveness for all operations [22].

**Usage example.** Consider an administrator (a privileged client) who wants to switch server  $s$  off and invokes `ChangeConfig( $\{-s\}$ )`. By liveness, `ChangeConfig` completes, and by properties (1) and (2), it returns an activated configuration  $C \supseteq \{-s\}$ . The activation of  $C$  expires all configurations that do not contain  $C$ , and in particular, those that do not include  $-s$ . Hence,  $s$  is not part of the membership of any unexpired configuration, and by the availability condition, the administrator can safely switch  $s$  off immediately once `ChangeConfig( $\{-s\}$ )` returns.

## 4 Reconfiguration Abstraction

We introduce a generic reconfiguration abstraction, which can be used for implementing dynamic objects as we illustrate in the next section. A Reconfiguration abstraction has two operations:

**Propose( $C, P$ )** for a configuration  $C$  and a proposed set of changes  $P$ ; and  
**Check( $C$ )** for a configuration  $C$ .

`Propose` is used to reconfigure the system, whereas `Check` is used in order to learn about other clients’ reconfiguration attempts. `Propose` and `Check` invoke the introduce and activate

events. Both Check and Propose return a pair of values  $\langle D, S \rangle$ , where  $D$  is a configuration and  $S$  is a *speculation set* containing configurations; when  $\langle D, S \rangle$  is returned we say that  $D$  is *nominated* by the operation that returns it. Intuitively, a nominated configuration is one that has been introduced and is a candidate for activation. By convention, we say that  $C_{init}$  is nominated at time 0. We assume that the first argument passed to both operations is a nominated configuration.

The first property of Reconfiguration is validity, which (i) requires  $\text{Propose}(C, P)$  to include  $P$  in the returned nominated configuration; and (ii) does not allow configurations to include spurious changes not proposed by any client. Formally:

$D_1$  (Validity) (i) If  $\text{Propose}(C, P)$  returns  $\langle D, S \rangle$  for some  $S$ , then  $D \supseteq P$ , and (ii) for every configuration  $D$  that is introduced or nominated by an operation  $op$ , for every  $e \in D \setminus C_{init}$ , there is a  $\text{Propose}(C', P')$  for some  $C'$  that is invoked before  $op$  returns s.t.  $e \in P'$ .

The second property ensures that nominated configuration sizes monotonically increase over time, which is essential for real-time order of operations invoked on objects that use this abstraction:

$D_2$  (Real-time Order) A configuration  $D$  nominated by operation  $op$  is larger than or equal to every configuration nominated by an operation that strictly precedes  $op$ .

Since Reconfiguration is weaker than consensus, clients do not agree on a sequence of nominated configurations. Hence, in case some client  $c_1$  proceeds to a configuration  $C'$ , we want to ensure that if another client  $c_2$  “skips”  $C'$ ,  $c_2$  has  $C'$  in its speculation set, and can thus transfer any state that  $c_1$  may have written there to the newer configuration  $c_2$  nominates. This is captured by property  $S_1$ (ii) below. Property  $S_1$ (i) stipulates that these configurations are also introduced, ensuring a live majority in these configurations in order to allow state transfer.

$S_1$  (Speculation) If  $\text{Check}(C)$  or  $\text{Propose}(C, P)$  returns  $\langle D, S \rangle$ , then every  $C' \in S$  is (i) introduced and (ii)  $S$  includes all nominated configurations  $C'$  s.t.  $|C| \leq |C'| \leq |D|$ . As a practical matter, if any  $C'$  between  $C$  and  $D$  has been activated, any  $C''$  s.t.  $|C''| < |C'|$  may be omitted.

In addition, we have to define when configurations are activated. Note that an activation of a new configuration leads to expiration of old ones, and thus to possible loss of information stored in them. Therefore, a configuration  $D$  is not immediately activated when a Propose returns  $\langle D, S \rangle$  for some  $S$ . Instead, a configuration  $C$  is activated if  $\text{Check}(C)$  does not report any newer configuration:

$A_1$  (Activation) If  $\text{Check}(C)$  returns  $\langle C, S \rangle$  for some  $S$ , then  $C$  is activated.

The liveness property of Reconfiguration is the same as in other dynamic objects [3, 18, 13, 23], namely, if the number of Propose operations is finite, then every operation by a correct client completes.

## 5 Building Dynamic Objects Using Reconfiguration

We first present a dynamic atomic read/write register emulation using Reconfiguration, and then explain the modifications needed for supporting a dynamic atomic max-register [4]. A formal proof is provided in Appendix A.4.

## 5.1 Dynamic atomic read/write register

Besides the Reconfiguration abstraction, our dynamic register implementation uses a (static) *ranked register* [11] emulation in every configuration, as illustrated in Figure 1a. A ranked register stores a tuple, called *version*, that consists of a value  $v$  and a monotonically increasing timestamp  $ts$ , and supports  $RRRead()$  and  $RRWrite(version)$  operations. The sequential specification of a *ranked register* is following: An  $RRRead()$  returns the version with the highest  $ts$  written by an  $RRWrite$  that precedes it, or  $\perp$  if there is no preceding  $RRWrite$ . Like all static objects in our model, if the configuration where the ranked register is emulated expires, the oracle returns an error.

The basic framework for implementing the Read, Write, and ChangeConfig operations is a loop: (i) Check, (ii) read (using  $RRread$ ) the highest version from all speculated configurations returned by Check, (iii) write (with  $RRWrite$ ) the highest version to the configuration nominated by Check, (iv) repeat. The loop terminates when Check does not nominate a new configuration. The specific action of each of the three operations is as follows. A Read simply returns the value of the highest version at the end of the loop. A Write increments the timestamp and writes it with a new value at the beginning of the loop. ChangeConfig proposes a configuration change via Propose instead of Check in the first iteration.

---

**Algorithm 1** Dynamic atomic read/write register using Reconfiguration.

---

**Client local variables:**

- 1: configuration  $C_{curr}$ , initially  $C_{init}$
- 2:  $TS = \mathbb{N} \times \Pi$  with selectors  $num$  and  $id$
- 3:  $version \in \mathbb{V} \times TS$  with selectors  $v$  and  $ts$ , initially  $\langle v_0, \langle 0, \text{client's id} \rangle \rangle$
- 4:  $pickTS \in \{true, false\}$ , initially  $true$ .

**Code for client  $c_i \in \Pi$ :**

- 5: **Read()**
- 6:    $transferState(Check(C_{curr}), \perp)$
- 7:    $checkConfig()$
- 8:   return  $version.v$
- 9: **Write(v)**
- 10:    $transferState(Check(C_{curr}), v)$
- 11:    $checkConfig()$
- 12:    $pickTS \leftarrow true$
- 13:   return ok
- 14: **ChangeConfig(P)**
- 15:    $transferState(Propose(C_{curr}, P), \perp)$
- 16:    $checkConfig()$
- 17:   return  $C_{curr}$
- 18: **On  $\langle error, D \rangle$  do**
- 19:    $C_{curr} \leftarrow D$
- 20:   restart operation
- 21: **procedure  $checkConfig()$**
- 22:    $\langle D, S \rangle \leftarrow Check(C_{curr})$
- 23:   **while**  $D! = C_{curr}$  **do**
- 24:      $transferState(\langle D, S \rangle, \perp)$
- 25:      $\langle D, S \rangle \leftarrow Check(C_{curr})$
- 26: **procedure  $transferState(\langle D, S \rangle, value)$**
- 27:   **for each**  $C \in S$  **do**
- 28:      $tmp \leftarrow C.RRRead()$
- 29:     **if**  $tmp.ts > version.ts$  **then**
- 30:        $version \leftarrow tmp$
- 31:     **if**  $value \neq \perp \vee pickTS = true$  **then**
- 32:        $version \leftarrow \langle value, \langle version.ts.num + 1, i \rangle \rangle$
- 33:      $pickTS \leftarrow false$
- 34:      $D.RRWrite(version)$
- 35:      $C_{curr} \leftarrow D$

---

The pseudocode appears in Algorithm 1. The  $transferState$  method reads the register's



version from the entire speculation set  $S$  and writes the latest version to the new configuration  $D$ . The *checkConfig* method repeatedly calls *transferState* until the configuration returned by *Check* stops changing. During the loop execution, an operation on an expired configuration may incur an exception, with a notification of the form  $\langle error, D \rangle$  (see line 18). In this case, the loop is aborted and the operation starts over at configuration  $D$ . In case write is restarted after it has chosen a new timestamp, it skips the timestamp selection step.

We say that a configuration  $C$  becomes *stable* when some version is written to  $C$  in step (iii). We refer to the first version written to  $C$  as the *opening* version of  $C$ . Consider a completed operation (Read, Write, or ChangeConfig)  $op$  and let  $C$  be the last configuration in which  $op$  writes some version  $v$ , we say that  $op$  commits  $v$  in  $C$  when it completes. The correctness of the register emulation, proven in Appendix A.4, is based on the following key invariant:

► **Invariant 1.** For every stable configuration  $C$ , the opening version of  $C$  is higher than or equal to the highest version committed in any configuration  $C'$  s.t.  $|C'| < |C|$ .

In other words, a larger stable configuration always holds a newer (or equal) version of the register's value than that committed in a smaller activated one.

---

**Algorithm 2** Dynamic atomic max-register using Reconfiguration.

---

**Client local variables:**

- 1: configuration  $C_{curr}$ , initially  $C_{init}$
- 2:  $value \in \mathbb{V}$ , initially  $v_0$

**Code** for client  $c_i \in \Pi$ :

<ol style="list-style-type: none"> <li>3: <b>MRead()</b></li> <li>4:   <math>transferState(Check(C_{curr}), \perp)</math></li> <li>5:   <math>checkConfig()</math></li> <li>6:   return <math>value</math></li>   <li>7: <b>MWrite(v)</b></li> <li>8:   <math>transferState(Check(C_{curr}), v)</math></li> <li>9:   <math>checkConfig()</math></li> <li>10:   return ok</li>   <li>11: <b>ChangeConfig(P)</b></li> <li>12:   <math>transferState(Propose(C_{curr}, P), \perp)</math></li> <li>13:   <math>checkConfig()</math></li> <li>14:   return <math>C_{curr}</math></li>   <li>15: <b>On</b> <math>\langle error, D \rangle</math> <b>do</b></li> <li>16:   <math>C_{curr} \leftarrow D</math></li> <li>17:   restart operation</li> </ol>	<ol style="list-style-type: none"> <li>18: <b>procedure</b> <math>checkConfig()</math></li> <li>19:   <math>\langle D, S \rangle \leftarrow Check(C_{curr})</math></li> <li>20:   <b>while</b> <math>D \neq C_{curr}</math> <b>do</b></li> <li>21:     <math>transferState(\langle D, S \rangle, \perp)</math></li> <li>22:     <math>\langle D, S \rangle \leftarrow Check(C_{curr})</math></li>   <li>23: <b>procedure</b> <math>transferState(\langle D, S \rangle, v)</math></li> <li>24:   <b>if</b> <math>v \neq \perp</math> <b>then</b></li> <li>25:     <math>value \leftarrow v</math></li> <li>26:   <b>for each</b> <math>C \in S</math> <b>do</b></li> <li>27:     <math>tmp \leftarrow C.MRead()</math></li> <li>28:     <b>if</b> <math>tmp &gt; value</math> <b>then</b></li> <li>29:       <math>value \leftarrow tmp</math></li> <li>30:   <math>D.MWrite(value)</math></li> <li>31:   <math>C_{curr} \leftarrow D</math></li> </ol>
--	--

---

**Complexity.** We measure complexity in terms of the number of accesses to low level objects, namely static atomic registers. Note that Read/Write/collect operations on static registers are emulated in a constant number of rounds using ABD. The complexity of the dynamic register's operations is determined by (1) the complexity of the operations inside the Checks invoked during the loop (plus possibly one Propose); and (2) the sum of the

sizes of all speculation sets returned by Propose/Check operations in this loop (where the register’s implementation performs Reads).

In a run with  $n$  ChangeConfig proposals, clearly, the best complexity we can hope for is  $O(n)$ . In the next section we present our algorithm for Reconfiguration, which achieves the asymptotically optimal  $O(n)$  complexity.

## 5.2 Dynamic atomic max-register

The emulation of a max-register on top of Reconfiguration is similar to the read/write register emulation. It differs in how we keep and transfer the state, i.e., the register’s value. First, instead of a (static) ranked register in each configuration, we use a (static) max-register. Second, instead of timestamps, we use the actual written values, that is, a writer writes its value in step (iii) only if it is higher than all the values read in step (ii) (Otherwise, it transfers the highest value it read). The pseudocode appears in Algorithm 2.

## 6 The Reconfiguration Abstraction Implementation

In this section we present an optimal and modular Reconfiguration implementation. In Section 6.1 we introduce the *Common Set (CoS)* building block, which is used by the Reconfiguration abstraction in every configuration. In Section 6.2 we show how CoS is used for non-optimal Reconfiguration and give the main correctness argument. In Section 6.3 we optimize the algorithm and give the main complexity and correctness claims. Formal proofs are deferred to Appendix A.

### 6.1 CoS building block

The *Common Set (CoS)* building block is a static shared object, emulated in every configuration  $C$  over a set of (static) registers. Its API consists of a single operation, denoted  $C.CoS(P)$ , where  $P$  is a set of arbitrary values.  $C.CoS$  returns an output set of sets satisfying the following:

► **Definition 2** (*Common Set in configuration  $C$* ).

- ( $CoS_1$ ) Each set in the output is the union of some of the inputs and strictly contains  $C$ ;
- ( $CoS_2$ ) if a client’s input strictly contains  $C$ , then its output is not empty;
- ( $CoS_3$ ) there is a common non-empty set in all non-empty outputs; and
- ( $CoS_4$ ) every  $C.CoS$  invocation that strictly follows a  $C.CoS$  call that returns a non-empty output returns a non-empty output.

For example, consider three concurrent clients that input to  $C.CoS$  the sets  $P_1$ ,  $P_2$ , and  $P_3$ , all of which contain  $C$ . A possible outcome is for their outputs to be  $\{P_1\}$ ,  $\{P_1, P_1 \cup P_2\}$ , and  $\{P_1, P_2, P_3\}$ , respectively. The intuitive explanation behind using CoS is that it builds a *common sequence* of configurations inductively: The first configuration in the sequence is  $C_{init}$ , the next is the common configuration returned by  $C_{init}.CoS$  (property  $CoS_3$ ), and so on. Although this sequence is not known to the clients themselves, every client observes this sequence starting with some activated configuration. Every configuration in this sequence contains the previous one.

CoS can be implemented directly using consensus or atomic snapshot, as illustrated in [24]. In Algorithm 3, (without the PreCompute function, which is an optimization and will be discussed later), we give an implementation based on DynaStore’s weak snapshot [3]. In the pseudocode, we denote by  $\bigcup S$  the union of all sets in a set of sets  $S$ . If the proposal

---

**Algorithm 3** Efficient CoS; algorithm of client  $p_i$  in configuration  $C$ ; optimization code shaded.

---

```

1: Local variables: ▷ flags accessible outside CoS
2:   firstTime set by reconfig and read by CoS
3:   drop set by CoS and read by reconfig

4: Shared variables (emulated in configuration  $C$ ):
5:   Boolean startingPoint, initially false ▷ Is  $C$  a starting point for some client
6:   Mapping from client to registers Warr and Sarr, initially {}.

7: procedure CoS( $P$ ) 16: procedure PRECOMPUTE( $P$ )
8:    $P \leftarrow PreCompute(P)$  ▷ optimization 17:   if firstTime then
9:   if  $P \supset C$  then 18:      $C.startingPoint \leftarrow true$ 
10:     ▷ Something new to propose 19:      $C.Sarr[i] \leftarrow P$ 
11:      $C.Warr[i] \leftarrow P$  20:      $drop \leftarrow false$ 
12:    $ret \leftarrow C.collect(Warr)$  21:     if  $\neg C.startingPoint$  then
13:   if  $ret = \{\}$  then 22:       return  $P$ 
14:     return  $ret$  ▷ repeat collect until  $P$  stops changing.
15:   else 23:      $drop \leftarrow true$ 
24:      $tmp \leftarrow \bigcup C.collect(Sarr)$ 
25:     while  $tmp \neq P$  do
26:        $P \leftarrow tmp$ 
27:        $tmp \leftarrow \bigcup C.collect(Sarr)$ 
28:     return  $P$ 

```

---

$P$  strictly contains  $C$ ,  $p_i$  has something new to propose and it writes  $P$  into its cell in the “weak” snapshot array  $Warr$  (lines 9-10). (Note that  $Warr$  is a static array emulated in the configuration where CoS is implemented). Either way, it collects  $Warr$  (line 11). In case the collect is not empty,  $p_i$  collects  $Warr$  again and returns the set of collected proposals (lines 12-15). The second collect ensures that the intersection of non-empty outputs includes the first written input, implying  $CoS_3$ ; the remaining properties are satisfied by a single collect.

## 6.2 Simple Reconfiguration

Given CoS, we can solve Reconfiguration in a generic way as shown in Algorithm 4 (ignore the shaded areas for now). Both Check and Propose use the auxiliary procedure *reconfig*. Propose( $C, P$ ) first sets a local variable *proposal* to the union of  $C$  and  $P$ , whereas Check( $C$ ) initiates *proposal* to be  $C$ . Both then execute the loop in line 40. Each iteration selects the smallest configuration in *ToTrack*; we say that the iteration *tracks* this configuration. The loop tracks all configurations returned by CoS, smallest to largest, starting with  $C$ . In each tracked configuration  $C'$ , the client introduces  $C'$ , invokes  $C'.CoS(proposal)$  and adds to *proposal* the union of the configurations returned from  $C'.CoS$ . This repeats for every configuration  $C'$  returned from CoS until there are no more configurations to track. Recall that by the liveness condition, if some configuration  $C'$  is expired and no longer supports  $C'.CoS$ , then the client gets in return to  $C'.CoS$  an exception with some newer activated configuration  $C_a$ . In this case, *reconfig* starts over from  $C_a$ . At the end, Propose and Check return *proposal* and the set of all tracked configurations.

The common sequence starts with  $C_{init}$ , and is inductively defined as follows: If  $C_k.CoS$  has a non-empty output, then  $C_{k+1}$  is the smallest common configuration returned by all non-empty  $C_k.CoS$ s. By  $CoS_3$ , all non-empty return values have at least one configuration in common, and if there is more than one such configuration, then we pick the smallest, breaking ties using lexicographic order. By  $CoS_1$ , each configuration in the common sequence

strictly contains the previous one.

---

**Algorithm 4** Generic Reconfiguration algorithm; optimization code shaded.

---

```

29: Propose( $C, P$ )
30:   return reconfig( $C, P$ )

31: Check( $C$ )
32:    $ret \leftarrow \text{reconfig}(C, \{\})$ 
33:   if  $ret = \langle C, * \rangle$  then activate( $C$ )
34:   return  $ret$ 

35: procedure reconfig( $C, P$ )
36:    $proposal \leftarrow P \cup C$ 
37:    $ToTrack \leftarrow \{C\}$ 
38:    $speculation \leftarrow \{\}$ 
39:    $firstTime \leftarrow true$ 
40:   while  $ToTrack \neq \{\}$  do
41:      $C' \leftarrow \underset{C'' \in ToTrack}{\text{argmin}} |C''|$      $\triangleright$  smallest
configuration
42:     introduce( $C'$ )
43:      $speculation \leftarrow speculation \cup \{C'\}$ 
44:      $ret \leftarrow C'.CoS(proposal)$ 
45:     if  $ret = \langle \text{"error"}, C_a \rangle$  then
46:        $\triangleright C'$  is expired - restart from  $C_a$ 
       return reconfig( $C_a, proposal$ )
47:      $ToTrack \leftarrow (ToTrack \cup ret) \setminus \{C'\}$ 
48:      $firstTime \leftarrow false$ 
49:     if  $drop = true$  then
50:        $\triangleright$  drop old configurations in  $ToTrack$ 
        $ToTrack \leftarrow ret$ 
51:      $proposal \leftarrow proposal \cup \bigcup ToTrack$ 
52:      $C_{curr} \leftarrow proposal$ 
53:     return  $\langle proposal, speculation \rangle$ 

```

---

By assumption,  $\tilde{C}_j \neq D_j$ , and thus,  $rec_j$  gets a non-empty output from  $\tilde{C}_j.CoS$  (it gets an output since we assume that there are no exceptions). But, this output includes some configuration in the common sequence, so  $rec_j$  tracks a configuration in the common sequence after  $\tilde{C}_j$ . A contradiction.

Liveness follows since (i) every call to CoS returns, either successfully or with an exception; and (ii) tracked configurations are monotonically increasing, and, provided that the number of reconfigurations is finite, they are bounded.

### 6.3 Optimal Reconfiguration

The key to the efficiency of our new algorithm is in its thrifty CoS implementation, and the signals it conveys to the reconfiguration algorithm, which minimize the number of tracked configurations. To this end, the efficient solution for CoS shares (local) state variables  $firstTime$  and  $drop$  with the Reconfiguration implementation.

To explain the intuition behind our algorithm, let us first consider a scenario in which all clients invoke register operations (Read, Write, or ChangeConfig) in the same starting

**Correctness.** The validity property ( $D_1$ ) immediately follows from CoS property  $CoS_1$  and the observation that *proposal* is set to include  $P$  at beginning of *reconfig* and never decreases.

To provide intuition for the remaining properties, we discuss the case in which all operations start in  $C_{init}$  and no exceptions occur; the proof for the general case appears in Appendix A.2. Observe that since *proposal* always contains  $\bigcup ToTrack$  and configurations are traversed from smallest to largest, we get from property  $CoS_2$  that  $C.CoS$  returns an empty set only if  $C$  includes  $ToTrack$ , i.e.,  $C$  is the last traversed configuration. The key correctness argument is that all nominated configurations belong to the common sequence, and are thus related by containment:

► **Lemma 3.** *For every reconfig that returns  $\langle D, S \rangle$ ,  $D$  belongs to the common sequence.*

*Proof - sketch for the special case (starting in  $C_{init}$ , no exceptions).* Assume by way of contradiction that  $D_j$  is returned by reconfig operation  $rec_j$  but does not belong to the common sequence. Note that  $C_{init}$  is in the common sequence and is tracked by  $rec_j$ . Let  $\tilde{C}_j$  be the last configuration tracked by  $rec_j$  that belongs to the common sequence.

configuration  $C_0$  (e.g.,  $C_0$  may be  $C_{init}$ ), and no exceptions occur. If  $n$  of the clients invoke Propose, then there are  $n$  sets  $P_1, \dots, P_n$  proposed by  $reconfig(C, P_i)$  operations. The unoptimized (weak snapshot-based) CoS may return up to  $2^n$  different subsets in CoS responses (assuming many clients invoke Read/Write operations), inducing high complexity.

Our algorithm reduces this complexity by running a pre-computation phase in *PreCompute*, which imposes a containment order on all configurations passed to, and hence returned from, CoS. This is done by running a variant of (strong) atomic snapshot [1] on all client proposals in configuration  $C_0$ . Specifically, each process writes its own proposal  $P$  (line 19) to the “strong” array *Sarr*, and then (lines 24-27) repeatedly collects the union of all *Sarr* cells into  $P$ , until  $P$  stops changing. Like an atomic snapshot, this ensures that all results of *PreCompute* are related by containment. Note, however, that unlike an atomic snapshot, the complexity of this pre-computation is linear in the number of *different* proposals written, rather than in the number of participating processes; if collect encounters a newly written value that does not change the union of written values, *PreCompute* returns. In case all operations start in  $C_0$ , there are no new proposals in other configurations, and so the containment order is preserved throughout the computation. This ensures that the number of different configurations tracked by all clients is at most  $n$ .

Next, we account for the case that clients invoke (or restart due to exceptions) their operations in different starting configurations. We have to identify configurations where some client starts, and run *PreCompute* in them too. To this end, we have clients signal (by raising the *startingPoint* flag) if a configuration is their starting point. Every client that later runs *C.CoS* sees this flag true, and executes the pre-computation. If a client  $p_i$  sees the flag false in *C.CoS*,  $p_i$  does not run the pre-computation. Nevertheless, since  $p_i$  checks the flag after writing its value to *Sarr*,  $p_i$ 's proposal is already in the array before new clients that start in this configuration perform their collects, and so  $p_i$ 's proposal is contained in theirs. Thus, at this new starting point, all clients obtain proposals that are related by containment among themselves.

The tricky part is that old proposals that were included in *ToTrack* before the new starting point are not necessarily ordered relative to ensuing proposals, as in the following scenario:

- Clients  $p_1$  and  $p_2$  start in  $C_0$  and propose  $C_0 \cup \{+a\}$  and  $C_0 \cup \{+b\}$ , respectively;  $p_1$  gets  $\{C_1\}$ , where  $C_1 = C_0 \cup \{+a\}$ , from  $C_0.CoS$  and  $p_2$  gets  $\{C_1, C_2\}$ , where  $C_2 = C_0 \cup \{+a, +b\}$ .
- Client  $p_1$  tracks  $C_1$ , gets an empty set from  $C_1.CoS$ , and activates it. Client  $p_3$  starts in  $C_1$ , (which is now activated), proposes  $C_3 = C_1 \cup \{+c\}$  in  $C_1.CoS$ , and gets  $\{C_3\}$ .
- Later,  $p_2$  tracks  $C_1$ , and gets  $C_3$  in  $C_1.CoS$ 's output. At this point  $p_2$ 's *ToTrack* contains  $C_2$  and  $C_3$ , neither of which contains the other.

To achieve linear complexity, we have clients *drop* all configurations previously returned from CoS at all the starting points they encounter. One subtle point is ensuring safety in the presence of such drops, and our proof of the general case of Lemma 3 in Appendix A.2 addresses this issue.

Intuitively, since the purpose of tracking all configurations is to ensure that clients traverse the common sequence, once we know  $C$  is in the common sequence, there is no need to continue to track any configuration older than  $C$ . So, the drop is safe.

A second subtle point is preserving linear complexity despite executing *PreCompute* in multiple starting points. But since (i) the worst-case complexity of a single pre-computation is linear in the number of different proposals written to it, (ii) each CoS begins with a proposal that reflects all those seen in previous CoSs, and (iii) there are  $n$  new proposals

overall, the combined complexity of *all* pre-computations is  $O(n)$ .

Finally, we provide intuition for the complexity of the high-level dynamic atomic register given in Section 5. The full proof, which wraps this intuition into a technical induction, appears in Appendix A.3. Recall that the register emulation performs a loop in which it repeatedly calls  $\text{Check}(C)$ , where  $C$  is the configuration returned from the previous Check/Propose, until some  $\text{Check}(C')$  returns  $\langle C', S \rangle$  for some  $C'$  and  $S$ . The loop performs a constant number of operations in every configuration returned in a speculated set  $S$  from Check. Therefore, we want the Checks in this loop to return the optimal number of configurations, and have optimal complexity themselves.

Since all the configurations introduced (and returned in speculation sets) by our algorithm are related by containment, we immediately conclude that the number of configurations returned in speculated sets  $S$  of all Checks together is bounded by  $n$ . Now we show that the complexity of all Checks combined is  $O(n)$ . First observe that all Checks combined invoke at most  $n$  CoSs. Second, each CoS writes at most three times to shared registers (lines 10, 18, and 19), reads once (in line 21), and performs each of the collects in lines 11, 15, and 24 at most once. Now observe that CoS performs the collect in line 27 only if the previous collect (in line 24 or 27) contained a proposal  $P_1 \not\subseteq P$ , which means that none of the CoSs collected  $P_1$  before. Since there are at most  $n$  proposals, all CoSs together perform the collect in line 27 at most  $n$  times. All in all, we get that the complexity of all Checks is  $O(n)$ .

## 7 Conclusions

We defined a dynamic model with a clean failure condition that allows an administrator to reconfigure an object and switch a removed server off once the reconfiguration operation completes. In this model, we have captured a succinct abstraction for consensus-less reconfiguration, which dynamic objects like atomic read/write register and max-register may use. We demonstrated the power of our abstraction by providing an optimal implementation of a dynamic register, which has better complexity than previous solutions in the same model.

---

### References

- 1 Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *Journal of the ACM (JACM)*, 40(4):873–890, 1993.
- 2 Marcos K. Aguilera, Idit Keidar, Dahlia Malkhi, Jean-Philippe Martin, and Alexander Shraer. Reconfiguring replicated atomic storage: A tutorial. *Bulletin of the EATCS*, 102:84–108, 2010. URL: <http://albcom.lsi.upc.edu/ojs/index.php/beatcs/article/view/43/48>.
- 3 Marcos K. Aguilera, Idit Keidar, Dahlia Malkhi, and Alexander Shraer. Dynamic atomic storage without consensus. *J. ACM*, 58(2):7:1–7:32, April 2011. URL: <http://doi.acm.org/10.1145/1944345.1944348>, doi:10.1145/1944345.1944348.
- 4 James Aspnes, Hagit Attiya, and Keren Censor. Max registers, counters, and monotone circuits. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, pages 36–45. ACM, 2009.
- 5 Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM (JACM)*, 42(1):124–142, 1995.
- 6 Hagit Attiya, Hyun Chul Chung, Faith Ellen, Saptarni Kumar, and Jennifer L Welch. Simulating a shared register in an asynchronous system that never stops changing. In *International Symposium on Distributed Computing*, pages 75–91. Springer, 2015.

- 7 Roberto Baldoni, Silvia Bonomi, and Michel Raynal. Implementing a regular register in an eventually synchronous distributed system prone to continuous churn. *Parallel and Distributed Systems, IEEE Transactions on*, 2012.
- 8 Ken Birman, Dahlia Malkhi, and Robbert Van Renesse. Virtually synchronous methodology for dynamic service replication. *Appears as Appendix A in [4]*, 2010.
- 9 Vita Bortnikov, Gregory Chockler, Dmitri Perelman, Alexey Roytman, Shlomit Shachor, and Ilya Shnayderman. Frappé: Fast replication platform for elastic services. *ACM LADIS*, 2011.
- 10 Gregory Chockler, Seth Gilbert, Vincent Gramoli, Peter M Musial, and Alex A Shvartsman. Reconfigurable distributed storage for dynamic networks. *Journal of Parallel and Distributed Computing*, 69(1):100–116, 2009.
- 11 Gregory Chockler and Dahlia Malkhi. Active disk paxos with infinitely many processes. *Distributed Computing*, 18(1):73–84, 2005.
- 12 Gregory Chockler and Alexander Spiegelman. The space hierarchy of fault tolerant register emulations. To appear at PODC 2017.
- 13 Eli Gafni and Dahlia Malkhi. Elastic configuration maintenance via a parsimonious speculating snapshot solution. In *Proceedings of the 29th International Symposium on Distributed Computing*, pages 140–153. Springer, 2015.
- 14 Seth Gilbert, Nancy Lynch, and Alex Shvartsman. RAMBO II: Rapidly reconfigurable atomic memory for dynamic networks. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 259–259. IEEE Computer Society, 2003.
- 15 Seth Gilbert, Nancy A Lynch, and Alexander A Shvartsman. RAMBO: A robust, reconfigurable atomic memory service for dynamic networks. *Distributed Computing*, 23(4):225–272, 2010.
- 16 Prasad Jayanti, Tushar Deepak Chandra, and Sam Toueg. Fault-tolerant wait-free shared objects. *J. ACM*, 45(3):451–500, May 1998. URL: <http://doi.acm.org/10.1145/278298.278305>, doi:10.1145/278298.278305.
- 17 Leander Jehl and Hein Meling. The case for reconfiguration without consensus. In *Proceedings of the 2016 ACM symposium on Principles of distributed computing*. ACM, 2016.
- 18 Leander Jehl, Roman Vitenberg, and Hein Meling. SmartMerge: A new approach to reconfiguration for atomic storage. In *Proceedings of the 29th International Symposium on Distributed Computing*, pages 154–169. Springer, 2015.
- 19 Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Reconfiguring a state machine. *SIGACT News*, 41(1):63–73, March 2010. URL: <http://doi.acm.org/10.1145/1753171.1753191>, doi:10.1145/1753171.1753191.
- 20 Nancy Lynch and Alex A Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *Distributed Computing*, pages 173–190. Springer, 2002.
- 21 Alexander Shraer, Jean-Philippe Martin, Dahlia Malkhi, and Idit Keidar. Data-centric reconfiguration with network-attached disks. In *Proceedings of the 4th International Workshop on Large Scale Distributed Systems and Middleware, LADIS '10*, pages 22–26, New York, NY, USA, 2010. ACM. URL: <http://doi.acm.org/10.1145/1859184.1859191>, doi:10.1145/1859184.1859191.
- 22 Alexander Spiegelman and Idit Keidar. On liveness of dynamic storage. *CoRR*, abs/1507.07086, 2015. URL: <http://arxiv.org/abs/1507.07086>.
- 23 Alexander Spiegelman and Idit Keidar. Dynamic atomic snapshots. In *Proceedings of the 2016 ACM symposium on Principles of distributed computing*. ACM, 2016.
- 24 Alexander Spiegelman, Idit Keidar, and Dahlia Malkhi. Dynamic reconfiguration: A tutorial. In *International Conference on Principles of Distributed Systems*, 2016.

## A Correctness Proofs

In Section A.1, we prove that our CoS algorithm (Algorithm 3) satisfies the required properties (Definition 2). In Section A.2, we prove that our Reconfiguration algorithm (Algorithm 4) on top of our CoS is correct, and in Section A.3 we prove that its complexity is optimal. In Section A.4 we prove the correctness of the dynamic atomic register emulation (Algorithm 1) on top of the Reconfiguration abstraction.

### A.1 SpSn Correctness

The proof makes use of the following simple observation:

► **Observation 1.** The output of *PreCompute* contains its input.

► **Lemma 4.** *Algorithm 3 implements CoS.*

**Proof.** We show the four properties of Definition 2:

*CoS<sub>1</sub>*. Each CoS's output is a set of proposals all of which were returned from *PreCompute* and, by line 9, strictly contain  $C$ . In *PreCompute*,  $P$  is always a subset of the union of inputs to CoS.

*CoS<sub>2</sub>*. Consider a configuration  $C$  and a client  $p_i$  that invokes  $C.CoS(P)$  s.t.  $P \supset C$ . By Observation 1,  $P \supset C$  also in line 9. Therefore,  $p_i$  writes  $P$  into  $Warr[i]$  (line 10), and since no other client writes into the same cell,  $p_i$  collects a non-empty set in its collect (lines 11 and 15). Thus,  $p_i$  returns a non-empty set.

*CoS<sub>3</sub>*. If there are no non-empty outputs, then we are done. Otherwise, there is at least one client that writes its  $P$  to *Warr*. Let  $p_i$  be the first such client (because *Warr* consists of atomic registers and atomicity is composable, the first is well-defined), and denote its  $P$  as  $P_i$ . We next show that every returned non-empty set contains  $P_i$ .

Consider a client  $p_j$  that returns a non-empty set. Then  $p_j$  collects *Warr* twice, and the first collect is not empty. Therefore,  $p_j$  completes its first collect after  $p_i$  writes  $P_i$  into  $Warr[i]$ , and thus, it is guaranteed that  $p_j$  reads  $P_i$  from  $Warr[i]$  during its second collect, and returns a set that contains  $P_i$ .

*CoS<sub>4</sub>*. Consider two complete C.CoS calls  $op_i$  and  $op_j$  s.t.  $op_i$  strictly follows  $op_j$ , and  $op_j$  returns a non-empty set. There is a non-empty cell in *Warr* before  $op_j$  completes, and since nothing is erased from *Warr*,  $op_i$ 's collect are not empty.

◀

### A.2 Reconfiguration correctness

**Notation.** We start with some notation that we use throughout the proof. The *common sequence* of configurations  $C_{init}, C_1, C_2, \dots$ , which only an outside observer can view, is inductively defined as follows: It starts with  $C_{init}$ , and if  $C_k.CoS$  has a non-empty output, then  $C_{k+1}$  is the smallest common configuration returned in all non-empty  $C_k.CoS$  outputs (by CoS property *CoS<sub>3</sub>*, the intersection is not empty), with ties broken in lexicographic order.

We say that a set is *monotonic* if all its elements are related by containment. A sequence is monotonic if every element  $C^k \neq C^0$  contains  $C^{k-1}$ . By CoS property *CoS<sub>1</sub>*, the common sequence is monotonic. For a *reconfig*( $C, P$ ) operation  $rec_j$ , we say a configuration is *tracked* by  $rec_j$  if it is selected as  $C'$  in line 41. We define *tracked*( $j$ ) =  $C_j^0, C_j^1, \dots, C_j^m$  to be the sequence of configurations tracked by  $rec_j$  in the last recursive call of *reconfig* in the order



they are tracked. Note that  $tracked(j)$  does not include configurations tracked before an exception is received. We further denote  $rec_j$ 's return value by  $\langle D_j, S_j \rangle$ .

The following observations follow immediately from the code:

► **Observation 2.** Consider a reconfig operation  $rec_j$ , then  $D_j = C_j^m \supseteq C_j^0 \cup \dots \cup C_j^m$ .

► **Observation 3.** If some reconfig reads  $drop=true$  in  $C.CoS$  at time  $t$ , then there is a reconfig  $rec_j$  s.t.  $C_j^0 = C$  that starts before time  $t$ .

The next claim stipulates that *reconfig* returns once CoS returns it an empty set.

► **Claim 4.** Consider a reconfig operation  $rec_j$  and a configuration  $C' \in tracked(j)$ . If  $C'.CoS$  called during  $rec_j$  returns an empty set, then  $C' = D_j$ .

**Proof.** By CoS property  $CoS_2$ , since  $C'.CoS$  returns an empty set, when  $C'.CoS$  is called (line 44),  $proposal \not\supseteq C'$ . Moreover, whenever CoS is called during a *reconfig* operation,  $proposal \supseteq \bigcup ToTrack$ . Together, we get that  $C'$  is not strictly contained in  $\bigcup ToTrack$ . Now notice that  $C'$  is selected in line 41 as  $\underset{C \in ToTrack}{\operatorname{argmin}} |C|$ . Therefore, we get that  $ToTrack = \{C'\}$

when  $C'.CoS$  is called (line 44). By the assumption,  $C'.CoS$  returns  $\{\}$ . Hence, in line:ToTrackUpdate1 50/47,  $ToTrack$  becomes  $\{\}$ , and thus the *reconfig* exits the while loop, and  $C' = D_j$ . ◀

The following observation follows from the usage of *reconfig* and the oracle behavior:

► **Observation 5.** For every reconfig operation  $rec_j$ ,  $C_j^0$  is a nominated configuration that is returned by some reconfig before  $rec_j$  is invoked.

The next claim shows that our algorithm drops old configurations only upon tracking a configuration in the common sequence.

► **Claim 6.** Assume that for every reconfig operation  $rec_j$  that returns before some time  $t$ ,  $D_j$  belongs to the common sequence. Now consider a configuration  $C$  that belongs to the common sequence, and a reconfig operation  $rec_i$  that tracks  $C$  and returns at time  $t$ . If  $rec_i$  gets a non-empty output from  $C.CoS$ , then  $rec_i$  tracks another configuration belonging to the common sequence after  $C$ .

**Proof.** First observe that  $rec_i$  gets a configuration  $C'$  that belongs to the common sequence from  $C.CoS$ , and  $rec_i$ 's  $ToTrack$  contains  $C'$  at the end of the corresponding while loop. If  $rec_i$  tracks  $C'$ , we are done. Otherwise,  $rec_i$  drops  $C'$  after calling some  $C''.CoS$  (while tracking  $C''$ ). By Observation 3, there is a *reconfig*  $rec_j$  s.t.  $C_j^0 = C''$  that starts before time  $t$ . By our assumption and by Observation 5,  $C''$  belongs to the common sequence, and we are done. ◀

We now show that every nominated configuration belongs to the common sequence.

**Lemma 3** (restated). *For every reconfig that returns  $\langle D, S \rangle$ ,  $D$  belongs to the common sequence.*

**Proof.** We prove by induction on time  $t \geq 0$  that for every *reconfig* operation  $rec_j$  that returns at time  $t$ ,  $D_j$  belongs to the common sequence.

**Base:** Since no operation returns at time 0 or earlier, the lemma holds for  $t = 0$ .

**Step:** We now assume that the lemma holds for some  $t \geq 0$ , and prove for  $t + 1$ . Let  $rec_j$  be a *reconfig* operation that returns at time  $t$ . By Observation 5,  $C_j^0$  was returned by some

*reconfig* before time  $t$ . Therefore, by the induction assumption  $C_j^0$  belongs to the common sequence. Now assume in a way of contradiction that  $D_j$  does not belong to the common sequence. Let  $C$  be the last configuration tracked by  $rec_j$  that belongs to the common sequence (there is such a configuration since  $C_j^0$  belongs to the common sequence). By the contradiction assumption,  $C \neq D_j$ , and thus by Claim 4,  $rec_j$  gets a non-empty output from  $C.CoS$ . Therefore, by Claim 6,  $rec_j$  tracks a configuration belonging to the common sequence after  $C$ . A get a contradiction. ◀

The following is an immediate conclusion from Lemma 3 and the monotonicity of the common sequence.

► **Corollary 5.** *The set of nominated configurations is monotonic.*

► **Claim 7.** Consider a  $reconfig(C, P)$  operation  $rec_j$  that returns  $D_j$ . Then  $rec_j$  tracks every configuration in the common sequence between  $C$  and  $D_j$ .

**Proof.** Assume by way of contradiction that there is a configuration  $C'$  in the common sequence between  $C$  and  $D_j$  that  $rec_j$  does not track. Let  $C''$  be the last configuration before  $C'$  in the common sequence that is tracked by  $rec_j$ . (There must be such a configuration because  $C = C_j^0$  is in the common sequence.) By Claim 4,  $rec_j$  gets a non-empty output from  $C''.CoS$ , and by the common sequence definition, this output contains the next configuration  $C^{next}$  in the common sequence. Now recall that  $rec_j$  tracks configurations from smallest to largest, and it does not track  $C^{next}$ . Therefore,  $rec_j$  drops  $C^{next}$  after calling  $CoS$  in some configuration  $\tilde{C}$  not in the common sequence. By Lemma 3,  $\tilde{C}$  is not nominated, and thus not activated. Therefore, by the oracle definition no *reconfig* starts in  $\tilde{C}$ , and thus the *drop* flag in  $\tilde{C}.CoS$  is always false. Hence,  $rec_j$  does not drop configurations after calling  $\tilde{C}.CoS$ . A contradiction. ◀

► **Theorem 6.** *Algorithms 4 and 3 implement the Reconfiguration abstraction.*

**Proof.** We now show that all Reconfiguration properties are satisfied:

- $D_1$  (i) Consider a  $Propose(C, P)$  operation that calls  $reconfig(C, P)$  and returns  $\langle D, S \rangle$ . We have to show that  $D \supseteq P$ . Now observe that *proposal* is set to contain  $P$  at the beginning of the *reconfig*, and never decreases, including recursive calls to *reconfig*. The property follows from line 53. (ii) Consider a *Check* or a *Propose* operation  $op$  that introduces, activates, or return configuration  $C'$ , and consider  $e \in C' \setminus C \cup P$ . Observe that  $op$  gets  $e$  by some  $CoS$  output. The property follows by inductively using  $CoS_1$ .
- $D_2$  Consider an operation (*Propose* or *Check*)  $op_j$  that strictly precedes another operation  $op_i$ . Let  $rec_j$  be the *reconfig* operation that is called during  $op_j$ , and let  $rec_i$  be the *reconfig* operation that is called during  $op_i$ . Notice that  $rec_j$  strictly precedes  $rec_i$ . By Lemma 3,  $D_j$  and  $D_i$  are in the common sequence. From  $CoS_1$ , either  $D_i \supseteq D_j$  or  $D_j \supseteq D_i$ . Assume by contradiction that  $|D_i| < |D_j|$ . Thus,  $D_j \supset D_i$ , and  $D_i$  precedes  $D_j$  in the common sequence. This means that  $D_i.CoS$  returned a non-empty output to some *reconfig* operation before  $D_j$  was added to the common sequence, and so before  $rec_j$  returned  $D_j$ . Since  $rec_j$  strictly precedes  $rec_i$ , we get that  $D_i.CoS$  returned a non-empty output before  $rec_i$  invoked  $D_i.CoS$ , and so by  $CoS_4$ ,  $D_i.CoS$  returns a non-empty output also to  $rec_i$ , a contradiction to the assumption that  $rec_i$  returns  $D_i$ .

$S_1$  Consider a  $\text{Check}(C)$  or  $\text{Propose}(C, P)$  operation  $op$  that returns  $\langle D, S \rangle$ . Let  $rec_i = \text{reconfig}(C', P')$  be the last  $\text{reconfig}$  operation that is called during  $op$ . By the oracle definition, by Lemma 3, and since every activate configuration is also nominated,  $\text{reconfig}(C, P)$  recursively calls  $\text{reconfig}(C_a, *)$  only if  $C_a$  is activated and nominated, and  $|C_a| > |C|$ . Thus, if  $C' \neq C$ , then  $C'$  is activated and belongs to the common sequence. By Lemma 3, all the nominated configurations are in the common sequence. Therefore, by Claim 7 and by the observation that the *speculation* set of  $rec_i$  is  $\text{tracked}(i)$ ,  $S$  includes all nominated configurations  $C''$  s.t.  $|C'| \leq |C''| \leq |D|$ .

◀

### A.3 Reconfiguration complexity

**Notation.** For every introduced configuration  $C$ :

1. We define  $\text{OldProps}(C)$  to be the proposals suggested in  $C.\text{CoS}$  by  $\text{reconfig}$  operations starting their traversals before  $C$ . That is,

$$\text{OldProps}(C) \triangleq \{P \mid \exists \text{reconfig that calls } C.\text{CoS}(P) \text{ while its flag } \text{firstTime} = \text{false}\}$$

2. Since by Corollary 5, the set of nominated configurations is monotonic, there is at most one nominated configuration of a given size. Thus, we can define  $\text{Pred}(C)$  to be the biggest nominated configuration  $C'$  s.t.  $|C'| < |C|$  for  $C \neq C_{\text{init}}$ . For completeness, we define  $\text{Pred}(C_{\text{init}}) \triangleq C_{\text{init}}$ .

For every nominated configuration  $C'$ :

1. All introduced configurations that have  $C'$  as their  $\text{Pred}$  are its successors:

$$\text{Successors}(C') \triangleq \{C'\} \cup \{C \text{ is introduced} \mid C' = \text{Pred}(C)\}, \text{ and}$$

$$\text{successors}^i(C) \triangleq \{C' \in \text{successors}(C) \mid |C'| \leq i\}.$$

2. All configurations that are in  $\text{ToTrack}$  of  $\text{reconfig}$  operations after tracking  $C'$  are potential successors of  $C'$ :

$$\text{PotentialSuccessors}(C') \triangleq \{C \mid \text{some } \text{reconfig} \text{ calls } C'.\text{CoS} \text{ in line 44 and}$$

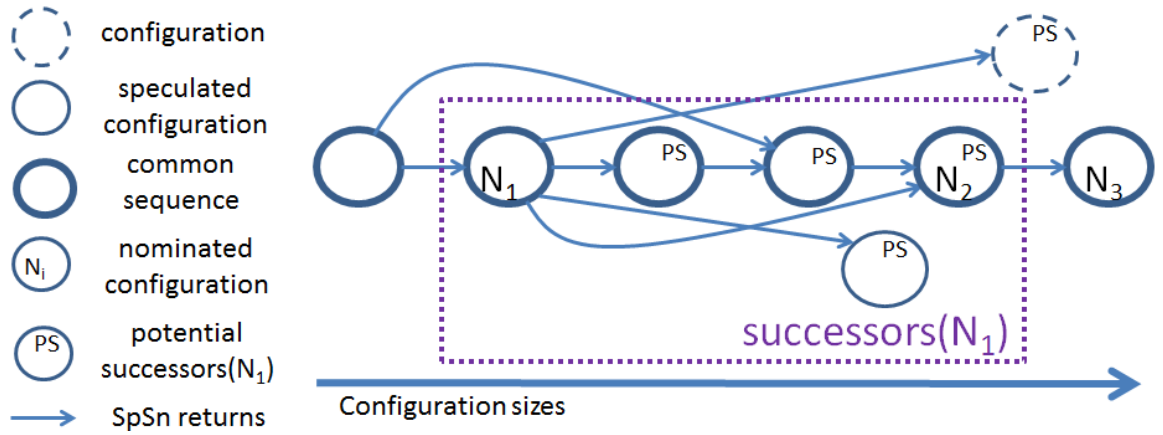
$$C \in \text{ToTrack} \text{ in line 51 in the same iteration}\}$$

An illustration of  $\text{Successors}$  and  $\text{PotentialSuccessors}$  appears in Figure 2.

We denote  $\text{introducedSet}^i \triangleq \{C \text{ is introduced} \mid |C| \leq i\}$ , and  $\text{nominatedSet}^i \triangleq \{C \text{ is nominated} \mid |C| \leq i\}$

We now state some observations that follow immediately from the code.

- ▶ **Observation 8.** Whenever  $\text{CoS}$  is called during a  $\text{reconfig}$  operation,  $\text{proposal} \supseteq \bigcup \text{ToTrack}$ .
- ▶ **Observation 9.** Consider a  $\text{reconfig}$  operation  $rec$  that calls  $C.\text{CoS}(P)$  s.t.  $P \supseteq C$  and returns  $ret$ . Then  $\exists C' \in ret$  s.t.  $P \subseteq C'$ .
- ▶ **Observation 10.** The return value of every  $\text{PreCompute}$  that reads  $\text{startPoint}=\text{false}$  in  $C.\text{CoS}$  is in  $\text{OldProps}(C)$ .
- ▶ **Observation 11.** If a configuration  $C'$  is returned by  $C.\text{CoS}$ , then there is a  $C.\text{CoS}$  invocation in which  $\text{PreCompute}$  returns  $C'$ .



■ **Figure 2** Example run of client  $p_i$  of our algorithm. The dashed configuration is in  $p_i$ 's *ToTrack* after calling  $N_1.CoS$ , and thus it is in  $potentialSuccessors(N_1)$ . But it is dropped after  $p_i$  calls  $N_2.CoS$ , and thus it is never introduced, and so it is not in  $successors(N_2)$ .

► **Corollary 7.** *If all reconfig operations that call  $C.CoS$  read  $startPoint=false$ , then all  $C.CoS$ 's return values in  $OldProps(C)$ .*

► **Observation 12.** Let  $P_1$  and  $P_2$  be two proposals returned by PreCompute executions during  $C.CoS$   $pc_1$  and  $pc_2$ , respectively. If  $pc_1$  and  $pc_2$  execute lines 24 to 27 (repeat collecting  $Sarr$  until  $P$  stops changing), then  $P_1$  and  $P_2$  are related by containment.

► **Claim 13.** Consider configuration  $C$ . If  $OldProps(C)$  is monotonic, then all configurations returned from  $C.CoS$  invocations are related by containment.

**Proof.** By Observation 11, we need to show that all proposals returned from PreCompute invoked during  $C.CoS$  are related by containment. Let  $P_1$  and  $P_2$  be two proposals returned by PreCompute executions during  $C.CoS$   $pc_1$  and  $pc_2$ , respectively. We show that  $P_1$  and  $P_2$  are related by containment. Consider three cases:

1. First,  $pc_1$  and  $pc_2$  read  $startPoint=false$  in line 21 executions during  $C.CoS$ . By Observation 10,  $P_1, P_2 \in OldProps(C)$ . Since by the assumption  $OldProps(C)$  is monotonic, we are done.
2. Second,  $pc_1$  reads  $startPoint=false$ , and  $pc_2$  reads  $startPoint=true$ . Note that since  $pc_1$  reads  $startPoint=false$ , it is called with  $P_1$ . Now since  $startPoint$  never changes from  $true$  to  $false$ ,  $pc_1$  reads  $startPoint$  before  $pc_2$ . Thus,  $pc_1$  writes  $P_1$  into  $Sarr$  in line 19 before  $pc_2$  reads  $startPoint=true$ . Therefore,  $pc_2$  sees  $P_1$  in all the collects in lines 24 to 27, and thus  $P_2 \supseteq P_1$ .
3. Third,  $pc_1$  and  $pc_2$  read  $startPoint=true$ . Therefore, both execute lines 24 to 27, and thus by Observation 12,  $P_1$  and  $P_2$  are related by containment.

◀

► **Claim 14.** If a configuration  $C$  is nominated, then  $C$  is introduced.

**Proof.** By Lemma 3,  $C$  belongs to the common sequence, and so by the common sequence definition  $C$  is introduced.

◀

► **Claim 15.** Consider a reconfig operation  $rec_j$  that introduces configuration  $C'$  with  $firstTime=true$  and later introduces  $C$ . Then,  $C' \subset C$ .

**Proof.** Note that when  $C'$  is introduced,  $ToTrack = \{C'\}$ . Recall that if an error is returned, then  $rec_j$  is aborted, and no later configurations are introduced by  $rec_j$ . Thus, no error is returned when  $rec_j$  introduces  $C'$ , and by CoS property  $CoS_1$ , at the end of the iteration all the configurations in  $ToTrack$  are strictly contain  $C'$ . Therefore, the claim follows by inductively repeating this argument. ◀

► **Corollary 8.** Consider an introduced configuration  $C$ . Then,  $C \supseteq C_{init}$ .

**Proof.** Let  $rec_j$  be a  $reconfig(C', P')$  that introduces  $C$ . Observe that  $rec_j$  introduces  $C'$  with  $firstTime=true$ . Thus, by Claim 15,  $C \supseteq C'$ . By Lemma 3,  $C'$  belongs to the common sequence, and by monotonicity of the common sequence,  $C' \supseteq C_{init}$ . Therefore,  $C \supseteq C_{init}$ . ◀

► **Claim 16.** Consider a  $reconfig$  operation  $rec_j$  that introduces configuration  $C$  in iteration  $iter$ . If  $firstTime=false$  at the beginning of  $iter$ , then there is at least one nominated configuration that is smaller than  $C$ , and  $rec_j$  tracks  $Pred(C)$  before  $iter$ .

**Proof.** Let  $C'$  be the configuration that  $rec_j$  is called with. Since  $C'$  and  $Pred(C)$  are nominated, by Corollary 5,  $C'$  and  $Pred(C)$  belong to the common sequence. Now observe that  $rec_j$  introduces  $C'$  with  $firstTime=true$  before it introduces  $C$ . Thus, by Claim 15,  $|C'| < |C|$ . If  $C' = Pred(C)$ , we are done, Otherwise,  $|C'| < |Pred(C)| < |C|$ . The Claim follows from Claim 7, and the observation that  $rec_j$  tracks configurations from the smallest to the biggest. ◀

► **Corollary 9.** Consider an introduced configuration  $C \neq C_{init}$ . Then there is at least one nominated configuration that is smaller than  $|C|$  and a reconfig operation that tracks  $Pred(C)$  and introduces  $C$ .

**Proof.** Let  $rec_j$  be a  $reconfig$  operation that introduces  $C$  in an iteration  $iter$ , and consider two cases:

1. First, flag  $firstTime=false$  at the beginning of  $iter$  during  $rec_j$ . The Corollary follows by Claim 16.
2. Otherwise,  $C$  is the parameter  $rec_j$  is called with, and thus  $C$  nominated. Let  $rec_i$  be the first  $reconfig$  that tracks  $C$ , and let  $t$  be that time. Therefore, no  $reconfig$  returns  $C$  before time  $t$ . hence,  $rec_i$  is not called with  $C$ , and thus  $firstTime=false$  when  $rec_i$  tracks  $C$ . The Corollary follows by Claim 16. ◀

► **Claim 17.** Consider a nominated configuration  $C$ , and an introduced configuration  $C^{i+1} \in successors(C)$  of size  $i + 1 > |C|$ . Assume that for every  $C' \in successors^i(C)$ ,  $PotentialSuccessors(C') \subseteq PotentialSuccessors(C)$ , and  $PotentialSuccessors(C)$  is monotonic. Then:

- (a)  $C^{i+1} \in PotentialSuccessors(C)$
- (b)  $OldProps(C^{i+1}) \subseteq PotentialSuccessors(C)$

**Proof.** (a) Since  $C^{i+1} \neq C_{init}$ , by Corollary 9, there is a *reconfig* operation  $rec_j$  that tracks  $C = Pred(C^{i+1})$  and introduces  $C^{i+1}$ . Now let  $C'^{i+1}$  be the biggest configuration in  $successors^i(C)$  that is tracked by  $rec_j$ . By the assumptions  $PotentialSuccessors(C'^{i+1}) \subseteq PotentialSuccessors(C)$ , and thus monotonic. Therefore, there is at most one configuration in  $PotentialSuccessors(C'^{i+1})$  whose size is  $i + 1$ . And since a *reconfig* operation tracks configurations by the order of their sizes, there is at least one configuration in  $PotentialSuccessors(C'^{i+1})$  whose size is  $i + 1$ . Therefore, there is exactly one configuration  $C''^{i+1}$  in  $PotentialSuccessors(C'^{i+1})$  of size is  $i + 1$ , and  $rec_j$  tracks  $C''^{i+1}$  immediately after  $C'^{i+2}$ . By CoS property  $CoS_1$ ,  $C''^{i+2} = C^{i+2}$ , and we are done.

(b) Consider some  $P \in OldProps_{C^{i+1}}$ , we need to show that  $P \in PotentialSuccessors(C)$ . Let  $rec_k$  be a *reconfig* operation that calls  $C^{i+1}.CoS$  while its flag *firstTime* = *false*. By Claim 16,  $rec_k$  tracks  $C$ . Let  $C'''^{i+1}$  be the biggest configuration in  $successors^i(C)$  that is tracked by  $rec_k$ . Since  $rec_k$  tracks configurations according to their sizes and since by (a)  $C^{i+1}$  is the only configuration in  $successors(C)$  of size is  $i + 1$ ,  $rec_k$  tracks  $C^{i+1}$  immediately after it tracks  $C'''^{i+1}$ . By Observation 9,  $rec_k$ 's proposal in  $C'''^{i+1}.CoS$  is included by some configuration in  $rec_k$ 's *ToTrack* before  $rec_k$  introduces  $C^{i+1}$ , and by definition,  $rec_k$ 's *ToTrack* is included in  $PotentialSuccessors(C'''^{i+1})$  before  $rec_k$  introduces  $C^{i+1}$ . Now By the assumptions,  $PotentialSuccessors(C'''^{i+1}) \subseteq PotentialSuccessors(C)$ , and thus monotonic. Therefore,  $P \in PotentialSuccessors(C'''^{i+1}) \subseteq PotentialSuccessors(C)$ , and we are done. ◀

► **Claim 18.** Consider a nominated configuration  $C$ . Assume that no more configurations of size  $|C|$  are introduced and  $\{C\} \cup PotentialSuccessors(C)$  is monotonic. Denote  $x \triangleq \max(\{j \mid \exists C' \in successors(C) : |C'| = j\})$ . Then for every  $|C| \leq i < x$ :

- For every  $C' \in successors^i(C)$ ,  $PotentialSuccessors(C') \subseteq PotentialSuccessors(C)$

**Proof.** We prove by induction on  $i$ .

**Base:**  $i = |C|$ . If  $i = x$ , we are done. Otherwise, by the assumption,  $C$  is the only introduced configuration of size  $|C|$ , and the lemma follows.

**Step:** Now assume that the lemma holds for some  $|C| \leq i < x$ , we show that it holds for  $i + 1$ . If  $i + 1 \geq x$ , we are done. If there is no configuration in  $successors(C)$  whose size is  $i + 1$ , then (1) follows by induction. Otherwise, by the Claim 17 (a) and since  $PotentialSuccessors(C)$  is monotonic, there is exactly one configuration  $C^{i+1} \in successors^{i+1}(C)$  of size is  $i + 1$ . Since  $i + 1 < x$ ,  $C^{i+1}$  is not nominated. Therefore, whenever  $C^{i+1}.CoS$  is called by a *reconfig* operation, its *firstTime* = *false*. Thus, all *reconfig* operations that call  $C^{i+1}.CoS$  read *startingPoint* = *false*. Now let  $C'^{i+1} \in PotentialSuccessors(C^{i+1})$ , we show that  $C'^{i+1} \in PotentialSuccessors(C)$ . If  $C'^{i+1}$  is returned by  $C^{i+1}.CoS$ , then by Corollary 7,  $C'^{i+1} \in OldProps(C^{i+1})$ . By Claim 17 (b),  $OldProps(C^{i+1}) \subseteq PotentialSuccessors(C)$ . Therefore,  $C'^{i+1} \in PotentialSuccessors(C)$ , and we are done. Otherwise, there is a *reconfig* operation  $rec$  that has  $C'^{i+1}$  in its *ToTrack* before it invokes  $C^{i+1}.CoS$ . Thus  $rec$ 's *firstTime* = *false* when it introduces  $C^{i+1}.CoS$ , and thus by Claim 16,  $rec$  track  $C$ . Now let  $C''^{i+1}$  be the last configuration  $rec$  tracks before  $C^{i+1}$ , and note that  $C''^{i+1} \in successors^i(C)$  and  $C'^{i+1} \in PotentialSuccessors(C''^{i+1})$ . By the induction assumption,  $PotentialSuccessors(C''^{i+1}) \subseteq PotentialSuccessors(C)$ . Therefore,  $C'^{i+1} \in PotentialSuccessors(C)$ , and we are done. ◀

The following corollary immediately follows from Claims 17 and 18:

► **Corollary 10.** Consider two nominated configurations  $C, C'$  s.t.  $C = \text{Pred}(C')$ , and assume that no more configurations with size  $|C|$  are introduced. If  $\{C\} \cup \text{PotentialSuccessors}(C)$  is monotonic, then (1)  $\text{successors}(C)$  is monotonic, (2)  $\text{OldProps}(C') \subseteq \text{PotentialSuccessors}(C)$ , and (3) for every  $C'' \in \text{successors}^{|C'|-1}(C)$ ,  $\text{PotentialSuccessors}(C'') \subseteq \text{PotentialSuccessors}(C)$ .

► **Claim 19.** Consider two nominated configurations  $C, C'$  s.t.  $C = \text{Pred}(C')$ , and let  $S$  be a monotonic set. If  $\text{OldProps}(C') \subseteq S$  and  $\forall C'' \in \text{successors}^{|C'|-1}(C)$ ,  $\text{PotentialSuccessors}(C'') \subseteq S$ , then  $\{C'\} \cup \text{PotentialSuccessors}(C')$  is monotonic.

**Proof.** Consider a configuration  $C_1 \in \text{PotentialSuccessors}(C')$ , we start by showing that  $C_1 \supset C'$ . By definition, there is a *reconfig*  $\text{rec}_1$  that calls  $C'.\text{CoS}$  in line 44 and  $C_1 \in \text{ToTrack}$  in line 51 in the same iteration. Now consider two cases:

1.  $C'.\text{CoS}$  in  $\text{rec}_1$  returns  $C_1$ . Therefore, by CoS property  $\text{CoS}_1$ ,  $C_1 \supset C'$ .
2.  $C_1$  is in  $\text{rec}_1$ 's *ToTrack* before it invokes  $C'.\text{CoS}$ . Thus,  $\text{rec}_1$  calls  $C'.\text{CoS}$  while its *firstTime* = *false*, and so by Claim 16,  $\text{rec}_1$  tracks  $C$ . Now let  $C''$  be the last configuration  $\text{rec}_1$  tracks before  $C'$ , and note that  $C', C_1 \in \text{PotentialSuccessors}(C'')$ . By the assumption,  $\text{PotentialSuccessors}(C'')$  is monotonic, and by the observation that  $\text{rec}_j$  tracks configurations from smallest to biggest,  $C_1 \supset C'$ .

Consider another configuration  $C_2 \in \text{PotentialSuccessors}(C')$ . By definition, there is a *reconfig*  $\text{rec}_2$  that calls  $C'.\text{CoS}$  in line 44 and  $C_2 \in \text{ToTrack}$  in line 51 in the same iteration. We now show that  $C_1$  and  $C_2$  are related by containment. there are three cases:

1. Both  $C_1$  and  $C_2$  are returned by  $C'.\text{CoS}$ . Therefore, by Claim 13,  $C_1$  and  $C_2$  are related by containment.
2. Neither  $C_1$  nor  $C_2$  is returned by  $C'.\text{CoS}$ . Thus  $C_1$  ( $C_2$ ) is in  $\text{rec}_1$ 's (respectively,  $\text{rec}_2$ 's) *ToTrack* before it invokes  $C'.\text{CoS}$ . Thus,  $\text{rec}_1$  and  $\text{rec}_2$  call  $C'.\text{CoS}$  while their *firstTime* = *false*, and so by Claim 16,  $\text{rec}_1$  and  $\text{rec}_2$  track  $C$ . Now let  $C'_1$  ( $C'_2$ ) be the last configuration  $\text{rec}_1$  (respectively,  $\text{rec}_2$ ) tracks before  $C'$ , and note that  $C_1 \in \text{PotentialSuccessors}(C'_1)$  (and  $C_2 \in \text{PotentialSuccessors}(C'_2)$ ). By the assumption,  $\text{PotentialSuccessors}(C'_1)$ ,  $\text{PotentialSuccessors}(C'_2) \subseteq S$ , and thus  $C_1, C_2 \in S$ . Now since  $S$  is monotonic, we are done.
3. One of the configurations, w.l.o.g.  $C_1$  is returned by  $C'.\text{CoS}$ , and  $C_2$  is not. In this case,  $C_2$  is in  $\text{rec}_2$ 's *ToTrack* before it invokes  $C'.\text{CoS}(P_2)$  and thus, by Observation 8,  $P_2 \supseteq C_2$ , and as in above  $C_2 \in S$ . In addition, since  $\text{rec}_2$  does not drop  $C_2$  after  $C'.\text{CoS}$  returns, it reads *startingPoint*=*false* during  $C'.\text{CoS}$ . By Observation 11, there is a *reconfig* operation  $\text{rec}'_1$  that gets  $C_1$  from *PreCompute* during  $C'.\text{CoS}$ . Now consider two cases:
  1.  $\text{rec}'_1$  reads *startingPoint*=*false*. Therefore,  $\text{rec}'_1$  calls  $C'.\text{CoS}$  while its *firstTime*=*false*, and *PreCompute* returns its input which is therefore  $C_1$ . Thus, by definition,  $C_1 \in \text{OldProps}(C')$ . By assumption,  $C_1 \in S$ , and we are done.
  2.  $\text{rec}'_1$  reads *startingPoint*=*true*. Since  $\text{rec}_2$  reads *startingPoint*=*false* during  $C'.\text{CoS}(P_2)$  and writes its proposal to *Sarr* (line 19) before reading *startingPoint* 21, and since *startingPoint* never changes from *true* *false* and  $\text{rec}'_1$  finds it true,  $\text{rec}'_1$  collects  $P_2$  in *Sarr* in line 24. Therefore,  $C_1 \supseteq P_2 \supseteq C_2$ .

◀

► **Claim 20.** The set  $\text{PotentialSuccessors}(C_{\text{init}}) \cup \{C_{\text{init}}\}$  is monotonic.

**Proof.** By Corollary 8 Claim 16, all *reconfig* operations that calls  $C_{init}.CoS$  do so with *first-Time=true*. Therefore,  $OldProps(C_{init}) = \{\}$  and all configurations in  $PotentialSuccessors(C_{init})$  are returned from  $C_{init}.CoS$ . Thus, by CoS property  $CoS_1$ , all configurations in  $PotentialSuccessors(C_{init})$  contain  $C_{init}$ , and by Claim 13, they are related by containment. The claim follows. ◀

► **Lemma 11.** *The set of introduced configurations is monotonic.*

**Proof.** Let  $x = |C_{init}|$ . We will show by induction on  $i \geq x$  that the following are satisfied:

- (a) The set  $introducedSet^i$  is monotonic.
- (b) For every configuration  $C \in nominatedSet^i$ ,  $\{C\} \cup PotentialSuccessors(C)$  is monotonic.

The lemma will follow from (a). **Base:** we prove for  $i = x$ . By Corollary 8, there is no introduced configuration other than  $C_{init}$  whose size is smaller than or equal to  $x$ . Hence, (a) is satisfied; (b) follows from Claim 20.

**Step:** assume by induction that (a) and (b) hold for  $i \geq x$ , we prove for  $i + 1$ . By Claim 14, every nominated configuration is also introduced, so if there is no introduced configuration whose size is  $i + 1$ , then we are done. Otherwise, let  $C$  be an introduced configuration s.t.  $|C| = i + 1$ , and let  $C_p = Pred(C)$ . Note that by definition,  $C \in successors(C_p)$ . Since  $|C| = i + 1 > x = |C_{init}|$ ,  $|C_p| < |C| = i + 1$ . Therefore, by the induction assumption (b),  $\{C_p\} \cup PotentialSuccessors(C_p)$  is monotonic, and by the induction assumption (a),  $C_p$  is the only introduced configuration of size  $|C_p|$ .

(a): By the first induction assumption  $introducedSet^{|C_p|}$  is monotonic, thus it is enough to show that  $C$  contains or equals every configuration  $C' \in introducedSet$  s.t.  $|C_p| \leq |C'| \leq i + 1$ . Note that, by definition,  $C' \in successors(C_p)$ . By Corollary 10 (1),  $successors(C_p)$  is monotonic. Therefore,  $C$  and  $C'$  are related by containment, and since  $|C| = i + 1 \geq |C'|$ ,  $C$  contains or equals  $C'$ , as requested.

(b): By (a),  $C$  is the only introduced configuration whose size is  $i + 1$ . If  $C$  is not nominated, then we are done. Otherwise, let  $S = PotentialSuccessors(C_p)$  by Corollary 10 (2),  $OldProps(C) \subseteq S$  and by 10 (3), for every  $C'' \in successors^{|C|-1}(C_p)$ ,  $PotentialSuccessors(C'') \subseteq S$ . Therefore, by Claim 19,  $\{C\} \cup PotentialSuccessors(C)$  is monotonic, as needed. ◀

We are now ready to conclude the complexity of the dynamic objects that uses our algorithm, which is captured by the following lemma:

► **Lemma 12.** *Consider an execution of a dynamic objects that uses our algorithm, and consider a loop in which  $Check(C)$  is repeatedly called, s.t.  $C$  is the configuration returned from the previous  $Check$ , until some  $Check(C')$  returns  $\langle C', * \rangle$ . Let  $n$  be the number of  $Propose(P)$  operations in the execution. Then:*

1. *All the Checks in the loop (together) return  $O(n)$  configurations in the speculations sets.*
2. *The complexity of all Checks in the loop combined is  $O(n)$ .*

**Proof.** Since every Check in the loop starts where the previous returns, the Checks in the loop introduce different configurations. Thus by Lemma 11, we immediately conclude that the number of configurations returned in speculated sets of all Checks in the loop together is bounded by  $n$ . Moreover, by  $CoS_1$ , no configuration is returned more than once in the



speculation sets. It is left to show that the complexity of all Checks combined is  $O(n)$ . First observe (again by Lemma 11) that all Checks combined invokes at most  $n$  CoSs. Second, each CoS writes at most three times to shared registers (lines 10, 18, and 19), reads once (in line 21), and performs each of the collects in lines 11, 15, and 24 at most once.

Now observe that CoS performs the collect in line 27 only if the previous collect (in line 24 or 27) contained a proposal  $P_1 \not\subseteq P$ , which means that none of the CoSs collected  $P_1$  before. Since there are at most  $n$  proposals, all CoSs together perform the collect in line 27 at most  $n$  times. All in all, we get that the complexity of all Checks in the loop is  $O(n)$ . ◀

#### A.4 Read/write register correctness

We prove here the correctness of the dynamic atomic read/write register appearing in algorithm 1.

**Notation.** We say that a version  $v$  is *higher* than a version  $v'$  if  $v.ts > v'.ts$ . An operation (Read, Write, or ChangeConfig) *stores* a version  $v$  in configuration  $C$  when it performs  $C.RRWrite(v)$  (line 34). A configuration  $C$  becomes *stable* when some operation stores a version in  $C$ . We say that the *opening* version of a stable configuration  $C$  is the first version that is stored in  $C$ . A configuration  $C$  *holds* a version  $v$  at time  $t$  if every  $C.RRRead()$  performed after time  $t$  returns a version that is higher than or equal to  $v$ . Consider a completed operation (Read, Write, or ChangeConfig)  $op$  and let  $C$  be the last configuration in which  $op$  stores some version  $v$ , we say that  $op$  *commits*  $v$  in  $C$  when it completes.

The following observation follows immediately from the specification of *ranked register*:

► **Observation 21.** A stable configuration holds at time  $t$  the highest version that was stored in it before time  $t$ .

The following observation follows immediately from the definitions of activated, stable and nominated configurations, and the code:

► **Observation 22.** Every activated configuration is Stable and every stable configuration is nominated.

The following observations follow immediately from the code:

► **Observation 23.** Every completed operation commits a version.

► **Observation 24.** A Check and Propose are always called with a stable configuration.

We now ready to prove the correctness of our register, which rely on the following invariant:

**Invariant 1** (restated). *For every stable configuration  $C$ , the opening version of  $C$  is higher than or equal to the highest version committed in any configuration  $C'$  s.t.  $|C'| < |C|$ .*

**Proof.** Assume in a way of contradiction that there is a time  $t$  at which the invariant does not hold. Let  $C_{st}$  be the smallest stable configuration that violate the invariant at time  $t$ , let  $v_{st}$  be the opening version of  $C_{st}$ , and let  $op_{st}$  be the operation that stores  $v_{st}$  in  $C_{st}$ . Now let  $t_1^{st}$  be the time when a *Check* or a *Propose* performed by  $op_{st}$  returns  $\langle C_{st}, S \rangle$  for some  $S$  for the first time. Denote this *Check* or *Propose* by  $CP_{st}$ , and let  $t_2^{st}$  be the time when  $op_{st}$  invokes procedure  $transferState(\langle C_{st}, S \rangle, \perp)$  (line 26). Note that  $t_1^{st} < t_2^{st} < t$ .

By the contradiction assumption there is at least one version, committed in a smaller configuration than  $C$  until time  $t$ , that is higher than  $v_{st}$ . Let  $op_c$  be an operation that

commits version  $v_c < v_{st}$  in configuration  $C_c$  until time  $t$  s.t.  $|C_c| < C_{st}$ . Now let  $t_1^c$  be the time when  $op_c$  stores  $v$  in  $C_c$  and let  $t_2^c > t_1^c$  be the time when  $op_c$  invokes  $Check(C_c)$  for the last time. Note that this  $Check(C_c)$  returns  $\langle C_c, S \rangle$  for some  $S$ . Now consider two case:

- First,  $t_1^{st} < t_2^c$ . Meaning that  $op_c$  invokes a  $Check(C)$  that nominates  $C$  after  $op_{st}$  nominates  $C_{st}$ . A contradiction to property  $D_2$  (Real time order) of the Reconfiguration abstraction.
- Second,  $t_1^{st} > t_2^c$ . Therefore,  $t_2^{st} > t_1^c$ , meaning that  $op_{st}$  performs  $transferState(\langle C_{st}, S \rangle, \perp)$  after  $op_c$  stores  $v$  in  $C_c$ . Let  $C_{in}$  be the input configuration to  $CP_{st}$ . Now consider two case:
  - First,  $C_c \in S$ . Since  $op_{st}$  performs  $C_c.RRRead()$  (in procedure  $transferState$ ) after  $t_1^c$ , and since  $transferState$  stores the highest version it reads, we get that  $v_{st}$  is higher than or equal to  $v_c$ . A contradiction.
  - Second,  $C_c \notin S$ . By property  $S_1$  (Speculation) of the Reconfiguration abstraction, this case is possible only if (1)  $|C_c| < |C_{in}| \leq |C_{st}|$  and  $C_{in} \in S$  or (2)  $S$  includes an activated configuration  $C_a$  s.t.  $|C_c| < |C_a| \leq |C_{st}|$ . By Observations 22 and 24,  $C_a$  and  $C_{in}$  are stable. Therefore, in both cases,  $S$  includes a stable configuration  $C'$  s.t.  $|C_c| < |C'| \leq |C_{st}|$ . Now consider two case:
    1. First,  $|C'| = |C_{st}|$ , meaning that  $C_{st}$  is stable at time  $t_1^{st}$ . Therefore, some operation stores a version in  $C_{st}$  before  $op_{st}$ . A contradiction to  $v_{st}$  being the opening version of  $C_{st}$ .
    2. Second,  $|C'| < |C_{st}|$ . Since  $C_{st}$  is the smallest stable configuration that violates the invariant at time  $t$ ,  $C_a$ 's opening version is higher than or equal to  $v_c$ . By Observation 21, and since  $transferState$  stores the highest version it reads from configurations in  $S$ , we get that  $v_{st}$  is higher than or equal to  $v_c$ . A contradiction.

◀

From Invariant 1 and Observation 21 we get the following corollary:

► **Corollary 13.** *At every time  $t$  a stable configuration  $C$  holds a version that is higher than or equal to the highest version committed in any configuration  $C'$  s.t.  $|C'| \leq |C|$  before time  $t$ .*

The following lemma follows from Corollary 13 and the specification of the reconfiguration abstraction:

► **Lemma 14.** *Consider an operation  $op_1$  that commits a version  $v_1$  in configuration  $C_1$ , and an operation  $op_2$  that begins after  $op_1$  returns. Let  $Check_2$  be a Check performed by  $op_2$ , and let  $\langle C_2, S_2 \rangle$  be the valued it returns. Then,  $S_2$  includes a stable configuration that holds a version higher than or equal to  $v_1$ .*

**Proof.** Note that  $op_1$  nominates  $C_1$ , and since  $op_1$  precedes  $op_2$ ,  $op_1$  performs a Check that nominates  $C_1$  before  $op_2$  invokes  $Check_2$ . Thus, by property  $D_2$  (real time order) of the reconfiguration abstraction,  $|C_1| \leq |C_2|$ . Moreover, by Observation 24 and property  $S_1$  (speculation) of the reconfiguration abstraction,  $S_2$  includes a stable configuration  $C_s$  s.t.  $|C_s| \geq |C_1|$ . By Corollary 13,  $C_s$  holds a version with a timestamp  $ts_s \geq ts_1$ .

◀

Notice that every completed Write operation picks exactly one timestamp (line 32), every picked timestamp is unique (ties are broken by clients ids), and there are no two different versions with the same timestamp. Thus, we say that Write operations are *associated* with the timestamp they pick. Note also that a Read operation returns the value in the version it commits. Therefore, we say that Read operations are associated with the timestamps of the versions they commit. The following Observation follows immediately from the code:

► **Observation 25.** A completed Write operation that is associated with timestamp  $ts$  commits a version with timestamp  $ts' \geq ts$ .

► **Lemma 15.** Consider two completed Write operations  $w_1, w_2$  that are associated with timestamps  $ts_1, ts_2$ , respectively. If  $w_1$  precedes  $w_2$ , then  $ts_1 < ts_2$ .

**Proof.** Let  $ts_1^c$  be the timestamp of the version committed by  $w_1$ . By Observation 25,  $ts_1^c \geq ts_1$ . Let  $Check_2$  be the last check  $w_2$  performs before picking a timestamp, and let  $\langle C_2, S_2 \rangle$  be its return value. By Lemma 14,  $S_2$  includes a stable configuration that holds a version with a timestamp  $ts_s \geq ts_1^c \geq ts_1$ . Therefore, by the code of *transferState*,  $ts_2 > ts_s \geq ts_1^c \geq ts_1$ . ◀

► **Lemma 16.** Consider two completed Read operations  $rd_1, rd_2$  that are associated with timestamps  $ts_1, ts_2$ , respectively. If  $rd_1$  precedes  $rd_2$ , then  $ts_1 \leq ts_2$ .

**Proof.** Let  $Check_2$  be the Check operation that  $rd_2$  performs before storing a version with timestamp  $ts_2$ , and let  $\langle C_2, S_2 \rangle$  be the return value of *check*<sub>2</sub>. By Lemma 14,  $S_2$  includes a stable configuration that holds a version with a timestamp  $ts_s \geq ts_1$ . Therefore, by the code of *transferState*,  $ts_2 \geq ts_s \geq ts_1$ . ◀

► **Lemma 17.** Consider a Write operation  $w_1$  associated with timestamp  $ts_1$ , and a Read operation  $rd_2$  associated with timestamp  $ts_2$ . If  $w_1$  precedes  $rd_2$ , then  $ts_1 \leq ts_2$ .

**Proof.** Let  $ts_1^c$  be the timestamp of the version committed by  $w_1$ . By Observation 25,  $ts_1^c \geq ts_1$ . Let  $Check_2$  be the Check operation that  $rd_2$  performs before storing a version with timestamp  $ts_2$ , and let  $\langle C_2, S_2 \rangle$  be the return value of *check*<sub>2</sub>. By Lemma 14,  $S_2$  includes a stable configuration that holds a version with a timestamp  $ts_s \geq ts_1^c \geq ts_1$ . Therefore, by the code of *transferState*,  $ts_2 \geq ts_s \geq ts_1^c \geq ts_1$ . ◀

► **Lemma 18.** Consider a Read operation  $rd_1$  associated with timestamp  $ts_1$ , and a Write operation  $w_2$  associated with timestamp  $ts_2$ . If  $rd_1$  precedes  $w_2$ , then  $ts_1 < ts_2$ .

**Proof.** Let  $Check_2$  be the last check  $w_2$  performs before picking a timestamp, and let  $\langle C_2, S_2 \rangle$  be its return value. By Lemma 14,  $S_2$  includes a stable configuration that holds a version with a timestamp  $ts_s \geq ts_1$ . Therefore, by the code of *transferState*,  $ts_2 > ts_s \geq ts_1$ . ◀

► **Definition 19** (linearization). For every run  $r$  we define the sequential run  $\sigma_r$  as follows: All the Write operations in  $r$  are ordered in  $\sigma_r$  by the timestamp they are associated with, and every Read operation associated with a timestamp  $ts$  in  $r$  is ordered in  $\sigma_r$  immediately after the Write that is associated with  $ts$ . Read operations that are associated with the same timestamps are ordered (among themselves) by the time they return.

## 23:28 Dynamic Reconfiguration: Abstraction and Optimal Asynchronous Solution

► **Theorem 20.** *The algorithm emulates an atomic register.*

**Proof.** We need to show that for every run  $r$ ,  $\sigma_r$  is a linearization of  $r$ . The sequential specification is satisfied by construction, and the real time order follows from Lemmas 15, 16, 17, and 18. ◀