Dynamic Reconfiguration: Abstraction and Optimal Asynchronous Solution*

Alexander Spiegelman[†1], Idit Keidar[2] and Dahlia Malkhi[3]

1 Viteri Dept. of Electrical Engineering, Technion, Haifa, Israel
2 Viterbi Dept. of Electrical Engineering, Technion, Haifa, Israel
3 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 $O(n)$.

1998 ACM Subject Classification F.1.2 Modes of Computation

Keywords and phrases Reconfiguration, Dynamic Objects, Optimal Algorithm

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.40

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 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

* A full version of the paper is available at `https://alexanderspiegelman.github.io/alexanderspiegelman.github.io/DynamicTasks.pdf`.

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

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).

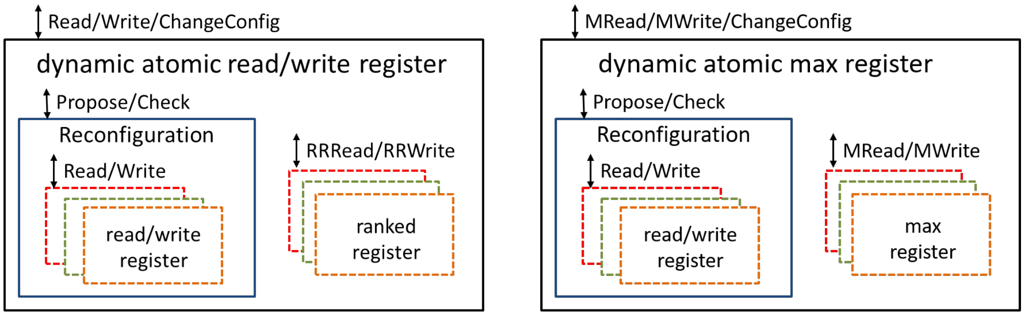
To this end , we first define a clean dynamic failure model, in which an administrator can immediately switch a server s off once a reconfiguration operation that removes s completes. Then, we provide an abstraction for consensus-less reconfiguration in this model. To demonstrate the power of our *Reconfiguration* 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[1] [12, 4]. Thus, a standalone implementation of dynamic max-registers is required.

Complexity. We present an optimal-complexity implementation of our Reconfiguration abstraction 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 as 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 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

[1] 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.

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 the full paper [25].

2 Related Work

Model. The problem of object reconfiguration has gained growing attention in recent years [15, 20, 3, 21, 18, 14, 24, 13, 23, 17, 22, 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 [23]. 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 [22]. 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 $\mathrm{O}(min(mn, 2^n))$ configurations, and requires a constant number of operations in every configuration, so $\mathrm{O}(min(mn, 2^n))$ 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 Π of *clients* (sometimes called processes), any number of which may fail by crashing, and an infinite set Φ 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:

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

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, 22]. 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 [23].

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 propert of Reconfiguration is validity, which (i) requires 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:

$\boldsymbol{D_1}$ (Validity) (i) If 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 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:

$\boldsymbol{D_2}$ (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.

$\boldsymbol{S_1}$ (Speculation) If Check(C) or 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 Check(C) does not report any newer configuration:

$\boldsymbol{A_1}$ (Activation) If 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, 22], 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 the full paper [25].

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

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:     $\mathit{transferState}(\mathit{Check}(C_{curr}), \perp)$
7:     $\mathit{checkConfig}()$
8:     return $\mathit{version}.v$

9: **Write(v)**
10:     $\mathit{transferState}(\mathit{Check}(C_{curr}), v)$
11:     $\mathit{checkConfig}()$
12:     $\mathit{pickTS} \leftarrow \mathit{true}$
13:     return ok

14: **ChangeConfig(P)**
15:     $\mathit{transferState}(\mathit{Propose}(C_{curr}, P), \perp)$
16:     $\mathit{checkConfig}()$
17:     return C_{curr}

18: **On** $\langle \mathit{error}, D\rangle$ **do**
19:     $C_{curr} \leftarrow D$
20:     restart operation

21: **procedure** $\mathit{checkConfig}()$
22:     $\langle D, S\rangle \leftarrow \mathit{Check}(C_{curr})$
23:     **while** $D! = C_{curr}$ **do**
24:         $\mathit{transferState}(\langle D, S\rangle, \perp)$
25:         $\langle D, S\rangle \leftarrow \mathit{Check}(C_{curr})$

26: **procedure** $\mathit{transferState}(\langle D, S\rangle, \mathit{value})$
27:     **for each** $C \in S$ **do**
28:         $\mathit{tmp} \leftarrow C.\mathit{RRRead}()$
29:         **if** $\mathit{tmp.ts} > \mathit{version.ts}$ **then**
30:             $\mathit{version} \leftarrow \mathit{tmp}$
31:     **if** $\mathit{value} \neq \perp \ \vee \ \mathit{pickTS} = \mathit{true}$ **then**
32:         $\mathit{version} \leftarrow \langle \mathit{value}, \langle \mathit{version.ts.num} + 1, i\rangle\rangle$
33:         $\mathit{pickTS} \leftarrow \mathit{false}$
34:     *D.RRWrite(version)*
35:     $C_{curr} \leftarrow D$

timestamp ts, and supports $\mathit{RRRead}()$ and $\mathit{RRWrite}(\mathit{version})$ operations. The sequential specification of a *ranked register* is following: An $\mathit{RRRead}()$ 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.

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 $\langle \mathit{error}, D\rangle$ (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.

Algorithm 2 Dynamic atomic max-register using Reconfiguration.

Client local variables:
configuration C_{curr}, initially C_{init}
$value \in \mathbb{V}$, initially v_0

Code for client $c_i \in \Pi$:

MRead()
  $transferState(Check(C_{curr}), \perp)$
  $checkConfig()$
  return $value$

MWrite(v)
  $transferState(Check(C_{curr}), v)$
  $checkConfig()$
  return ok

ChangeConfig(P)
  $transferState(Propose(C_{curr}, P), \perp)$
  $checkConfig()$
  return C_{curr}

On $\langle error, D\rangle$ **do**
  $C_{curr} \leftarrow D$
  restart operation

procedure $checkConfig()$
  $\langle D, S\rangle \leftarrow Check(C_{curr})$
  while $D! = C_{curr}$ **do**
    $transferState(\langle D, S\rangle, \perp)$
    $\langle D, S\rangle \leftarrow Check(C_{curr})$

procedure $transferState(\langle D, S\rangle, v)$
  if $v \neq \perp$ **then**
    $value \leftarrow v$
  for each $C \in S$ **do**
    $tmp \leftarrow C.MRead()$
    if $tmp > value$ **then**
      $value \leftarrow tmp$
  $D.MWrite(value)$
  $C_{curr} \leftarrow D$

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 the full paper [25], is based on the following key invariant:

▶ **Invariant 2.** 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.

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 can be found in the full paper [25].

6.1 CoS building block

The *Common Set (CoS)* building clock 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 3** (*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 P1, P2, and P3, all of which contain C. A possible outcome is for their outputs to be $\{P1\}, \{P1, P1 \cup P2\}$, and $\{P1, P2, P3\}$, 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 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.

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

```
Local variables:                                    ▷ flags accessible outside CoS
    firstTime set by reconfig and read by CoS
    drop set by CoS and read by reconfig

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

procedure CoS(P)
   P ← PreCompute(P)               ▷ optimization
   if P ⊃ C then
                    ▷ Something new to propose
      C.Warr[i] ← P
   ret ← C.collect(Warr)
   if ret = {} then
      return ret
   else
      return C.collect(Warr)

procedure PreCompute(P)
   if firstTime then
      C.startingPoint ← true
   C.Sarr[i] ← P
   drop ← false
   if ¬C.startingPoint then
      return P
   ▷ repeat collect until P stops changing.
   drop ← true
   tmp ← ⋃C.collect(Sarr)
   while tmp ≠ P do
      P ← tmp
      tmp ← ⋃C.collect(Sarr)
   return P
```

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.

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 the full paper [25]. Observe that since *proposal* always contains $\bigcup$ *ToTrack* and configurations

Algorithm 4 Generic Reconfiguration algorithm; optimization code shaded.

Propose(C, P)
  return $\mathit{reconfig}(C, P)$

Check(C)
  $\mathit{ret} \leftarrow \mathit{reconfig}(C, \{\})$
  if $\mathit{ret} = \langle C, * \rangle$ **then** $\mathit{activate}(C)$
  return ret

procedure $\mathit{reconfig}(C, P)$
  $\mathit{proposal} \leftarrow P \cup C$
  $\mathit{ToTrack} \leftarrow \{C\}$
  $\mathit{speculation} \leftarrow \{\}$
  $\mathit{firstTime} \leftarrow \mathit{true}$
  while $\mathit{ToTrack} \neq \{\}$ **do**
    $C' \leftarrow \underset{C'' \in ToTrack}{\operatorname{argmin}} |C''|$ ▷ smallest configuration
    $\mathit{introduce}(C')$
    $\mathit{speculation} \leftarrow \mathit{speculation} \cup \{C'\}$
    $\mathit{ret} \leftarrow C'.\mathit{CoS}(\mathit{proposal})$
    if $\mathit{ret} = \langle \text{“error”}, C_a \rangle$ **then** ▷ C' is expired - restart from C_a
      return $\mathit{reconfig}(C_a, \mathit{proposal})$
    $\mathit{ToTrack} \leftarrow (\mathit{ToTrack} \cup \mathit{ret}) \setminus \{C'\}$
    $\mathit{firstTime} \leftarrow \mathit{false}$
    if $\mathit{drop} = \mathit{true}$ **then** ▷ drop old configurations in *ToTrack*
      $\mathit{ToTrack} \leftarrow \mathit{ret}$
    $\mathit{proposal} \leftarrow \mathit{proposal} \cup \bigcup \mathit{ToTrack}$
  $C_{curr} \leftarrow \mathit{proposal}$
  return $\langle \mathit{proposal}, \mathit{speculation} \rangle$

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 4.** *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. 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 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 $P_1, \ldots, P_n$ 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 4 in the full paper [25] 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 the full paper [25]. Recall that the register emulation performs a loop in which it repeatedly calls Check(C), where C is the configuration returned from the previous Check/Propose, until some 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 $P_1 \nsubseteq P$, 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.

Acknowledgements. We thank Christian Cachin for helpful comments on our OPODIS 2015 tutorial, which served as the basis for parts of this paper. We thank Eli Gafni for interesting discussions.

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*, 2010.

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.

4 James Aspnes, Hagit Attiya, and Keren Censor. Max registers, counters, and monotone circuits. In *PODC 2009*, 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, Saptaparni 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. *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. Space complexity of fault-tolerant register emulations. In *Proceedings of PODC'17*. ACM, 2017.

13 Eli Gafni and Dahlia Malkhi. Elastic configuration maintenance via a parsimonious speculating snapshot solution. In *DISC 2015*, pages 140–153. Springer, 2015.

14 Seth Gilbert, Nancy Lynch, and Alex Shvartsman. RAMBO II: Rapidly reconfigurable atomic memory for dynamic networks. In *DSN 2013*. 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), 2010.

16 Prasad Jayanti, Tushar Deepak Chandra, and Sam Toueg. Fault-tolerant wait-free shared objects. *J. ACM*, 45(3):451–500, May 1998.

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 DISC 2015*. Springer, 2015.

19 Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Reconfiguring a state machine. *SIGACT News*, 41(1):63–73, March 2010.

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 *LADIS*. ACM, 2010.

22 Alexander Spiegelman and Idit Keidar. Dynamic atomic snapshots. In *Proceedings of the 2016 ACM symposium on Principles of distributed computing*. ACM, 2016.

23 Alexander Spiegelman and Idit Keidar. On liveness of dynamic storage. In *Proceedings of SIROCCO 2017*, 2017.

24 Alexander Spiegelman, Idit Keidar, and Dahlia Malkhi. Dynamic reconfiguration: A tutorial. In *International Conference on Principles of Distributed Systems*, 2016.

25 Alexander Spiegelman, Idit Keidar, and Dahlia Malkhi. Dynamic reconfiguration: Abstraction and optimal asynchronous solution, 2017. URL: `https://alexanderspiegelman.github.io/alexanderspiegelman.github.io/DynamicTasks.pdf`.