Dynamic Reconfiguration: A Tutorial*

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

1 **Andrew and Erna Viterbi Dept. of Electrical Engineering, Technion, Haifa, 32000, Israel**
sashas@tx.technion.ac.il, idish@ee.technion.ac.il

2 **VMware, Palo Alto, USA**
dahliamalkhi@gmail.com

Abstract

A key challenge for distributed systems is the problem of *reconfiguration*. Clearly, any production storage system that provides data reliability and availability for long periods *must* be able to reconfigure in order to remove failed or old servers and add healthy or new ones. This is far from trivial since we do not want the reconfiguration management to be centralized or cause a system shutdown.

In this tutorial we look into existing reconfigurable storage algorithms [7, 8, 1, 9, 6, 10]. We propose a common model and failure condition capturing their guarantees. We define a reconfiguration problem around which dynamic object solutions may be designed. To demonstrate its strength, we use it to implement dynamic atomic storage. We present a generic framework for solving the reconfiguration problem, show how to recast existing algorithms in terms of this framework, and compare among them.

1998 ACM Subject Classification C.2.4 Distributed Systems

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

1 Introduction

A key challenge for distributed systems is the problem of *reconfiguration*, i.e., changing the active set of servers. Clearly, any production system that provides data reliability and availability for long periods *must* be able to reconfigure in order to remove failed or old servers and add healthy or new ones. The foundations of reconfigurable distributed algorithms are key to understanding and designing dynamic distributed systems.

The study of reconfigurable replication has been active since at least the early 1980s, with the development of group communication and virtual synchrony (see survey in [3]). In recent years, there were several works on reconfigurable (dynamic) storage [7, 8, 1, 9, 6, 10], some of which use consensus for reconfigurations [7, 8] while others assume fully asynchronous systems [1, 9, 6, 10]. We feel that the time has come to provide a clear, unifying failure model and a framework for studying the relationship among different solutions.

In this tutorial we define a clear model, and a generic reconfiguration abstraction that can be used as a black-box in dynamic object emulations. In our model, a configuration is defined by sets of changes (such as adding and removing servers). The sequential specification of our reconfiguration problem says that there is a global sequence of configurations, totally ordered in a way that will be defined below. Importantly, it does not require that clients learn every configuration in the sequence, hence it does not necessitate (or imply) consensus.

* This work is partially suported by the Israeli Science Foundation. Alexander Spiegelman is grateful to the Azrieli Foundation for the award of an Azrieli Fellowship.

Conference title on which this volume is based on.
Editors: Billy Editor and Bill Editors; pp. 1–15

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Despite this weak guarantee, we demonstrate the usefulness of our reconfiguration abstraction by implementing a dynamic register on top of it. We also define a failure condition that generalizes the correct majority (of servers) condition from static systems to dynamic ones. On the one hand, our condition is strong enough to be useful, in that we allow servers to fail (or to be switched off) immediately when an operation that removes them from the current configuration returns. And on the other hand, it is sufficiently weak as to allow implementations that preserve the objects' states when the system is reconfigured.

We present a solution for the reconfiguration problem, which is based on the core mechanism in DynaStore [1]. In order to make it generic and simple, we define a *Speculating Snapshot* (SpSn) abstraction, based on [6], which is the core task clients have to solve in order to coordinate. We then show how to recast existing dynamic storage algorithms [7, 8, 1, 9, 6, 10] in terms of this framework. Specifically, we show that the SpSn abstraction can be implemented by extracting the core coordination mechanism from each of these algorithms, (e.g., consensus from RAMBO [7, 8] and weak snapshot from DynaStore [1, 10]). We use this unified presentation to compare their properties and the resulting complexity of reconfiguration.

The remainder of this tutorial is organized as follows: In Section 2 we define the model and failure condition. In Section 3 we define the reconfiguration problem. Then, in Section 4, we introduce the SpSn abstraction, present our generic reconfiguration algorithm, and compare among different SpSn implementations. In Section 5 we demonstrate how to use the reconfiguration algorithm in order to implement a dynamic atomic register. Finally, we conclude in Section 6.

2 Model

A dynamic shared storage system consists of an infinite set Φ of object servers supporting atomic *read-modify-write* (RMW) remote calls by an infinite set Π of clients. Calls may take arbitrarily long to arrive and complete, hence the system is asynchronous. Any number of clients may fail by crashing. The servers may also fail by crashing, but their failures are restricted by the failure model, which we define later. A server or client is *correct* in a run r if it does not fail in r, and otherwise, it is *faulty*.

We study algorithms that emulate reliable shared objects for the clients via dynamic subsets of the servers.

2.1 Configurations

Our definition of configurations is based on [1]; here we extend it to client-server systems. Intuitively, a configuration is a set of included or excluded servers. Fomally, we define *Changes* to be the set $\{+,-\} \times \Phi$. For simplicity we refer to $\langle +, s\rangle$ as $+s$ (and accordingly to $\langle -, s\rangle$ as $-s$). For example, $+s_3$ is a change that denotes the inclusion of server s_3. A *configuration* is a finite subset of Changes, e.g., $\{+s_1, +s_2 - s_2$, and $+s_3\rangle\}$ is a configuration representing the inclusion of servers s_1, s_2, and s_3, and the exclusion of s_2. For every configuration C, the membership of C, denoted $C.membership$, is the set of servers that are included but not excluded from it: $\{s| + s \in C \wedge -s \notin C\}$. An excluded server cannot be included again later; in practice, it might join with a different identity. Illustrations of a configuration and its membership appear in Figure 1. Tracking excluded servers in addition to the configuration's membership is important in order to reconcile configurations suggested by different clients. An initial configuration C_0 is known to all clients.

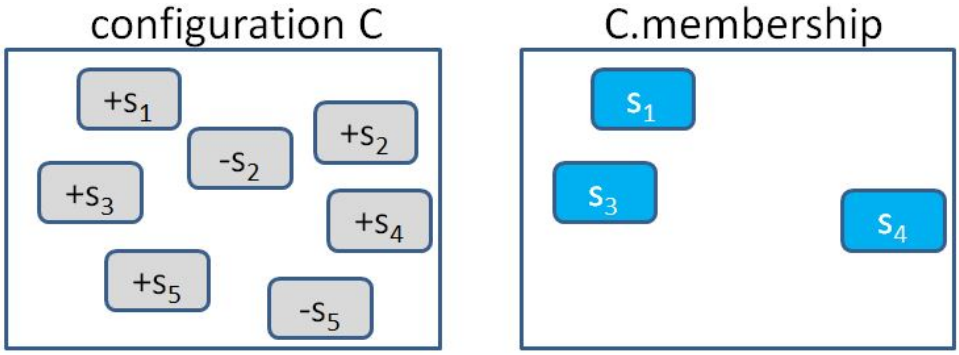

Figure 1 A configuration and its membership.

Configuration life-cycle At different times, configurations can be *speculated*, *activated*, and *expired*. A client can *speculate* a configuration C by issuing an explicit *speculate(C)* event, while configurations are activated and expired implicitly, as we later discuss. A configuration begins its life cycle with its speculation, which occurs after the inclusions and exclusions comprising it have been requested by clients. Then, if it "succeeds", it becomes activated when the system in some sense chooses to make it the new configuration, as we explain in Section 3 below. The newly activated configuration must contain all previously activated ones. (A server is removed by keeping its inclusion and adding its exclusion to the configuration.) A configuration is expired, whether or not it was activated, when a "newer" configuration is activated. Here, we refer to any configuration D as "newer" than C when C does not contain D. The activation of D prevents the future activation of C. Formally:

▶ **Definition 1** (life-cycle)**.** The lifecycle of a configuration C is defined by the following events:

speculate(C): An event that is invoked explicitly by clients.

activate(C): An event that is triggered automatically by certain client operations, as defined below.

expire(C): An event that occurs automatically and immediately when some configuration $C' \not\subseteq C$ is activated.

We later use these notions in order to define the failure model as well as the reconfiguration problem.

Shared register emulation For every configuration C, as long as a majority of $C.membership$ is alive, clients can use ABD [2] to simulate a collection of atomic read/writer registers on top of the servers in $C.membership$. Thus, we define:

▶ **Definition 2** (availability)**.** A configuration C is *available* if majority of the servers in $C.membership$ are correct.

For simplicity, we henceforth use this abstraction, and have clients invoke atomic read/write operations on shared registers in a given configuration. Note that if a configuration is unavailable, pending reads and writes to and from the configuration's registers might never return. We next define a failure condition that specifies which configurations must be available.

Failure condition There are infinity many servers in the system, and they cannot all be "alive" from the beginning. A *speculate* event indicates when we expect a configuration to become available. And now, the question is when a configuration can become unavailable. In our failure model we require *reconfigurablity*, which means that once we succeed to activate a new configuration C, every server s that is excluded in C (i.e., $-s \in C$) can fail or be switched off. To this end, we define the following failure condition, which is sufficiently weak so as to allow reconfigurablity.

▶ **Definition 3** (failure condition)**.** If configuration C is speculated and not expired, then C is available.

Recall that when a configuration C is activated then every configuration $D \not\supseteq C$ is expired. Intuitively, we can think of the activated configuration as *chosen*, of activated ones that are expired as *replaced*, and of speculated and not activated ones that are expired as *abandoned*. Figure 2 shows how our failure condition satisfies reconfigurablity.

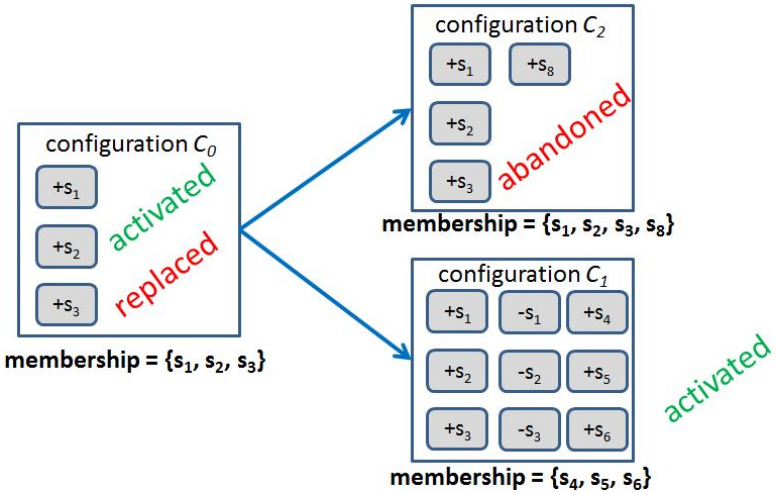

Figure 2 Example of activation and expiration. First client c_a speculates configuration C_1 in C_0 , while client c_b speculates configuration C_2 in C_0, client c_a misses C_2 and activates C_1. Note that $C_0, C_2 \not\supseteq C_1$, and thus, according to our definition, both C_0, C_2 are expired and are allowed to be unavailable. Therefore, the excluded servers in C_1 (s_1, s_2, and s_3) are no longer needed for liveness and can be safely switched off.

2.2 Discovering available configurations

Since configurations can be expired and become unavailable, we cannot guarantee termination of operations on emulated registers in expired configurations. Moreover, clients trying to access the shared object would need to access newer configurations in order to complete their operations. For example, a client that arrives after C_0 is expired and does not know of any newer configuration may hang forever because C_0 can already be unavailable. It is easy to see that this limitation is inherent in every model that separates clients from servers and requires reconfigurability (or any other failure model that allows failures of servers in an old configuration).

Therefore, clients have to somehow be notified about new activated configurations. Note that a speculated configuration C can become unavailable only when some configuration

$C' \not\subseteq C$ is activated. Thus, when a client tries to access an unavailable configuration C, we want to help the client find an activated configuration.

To this end, we envision that the system would use some *directory* service, or *oracle*, that stores information about configurations and ensures only a relaxed consistency notion[1]. A client which activates a configuration C informs the directory that C is activated. A client which tries to access a configuration D simultaneously posts queries to the directory about D. If the directory service sees that a configuration C has been activated such that D does not contain C, it reports back to the client that D has been expired by an activated configuration C. It is important to note that two clients that post queries to the directory about D can get different responses about activated configurations, and we do not require any eventual consistency properties on these reports. Hence, this service is weak and does not provide consensus.

Such a directory service can be easily implemented in a distributed manner with multiple directory servers. A client informs its local directory server about new activated configurations, and the server broadcasts it to all the other directory servers. When a client posts a request about an expired configuration D to its local directory server, then eventually the server will learn about some configuration that could have expired D and return it to the client.

In order to keep the model simple, we avoid using an explicit directory. Instead, we take an abstraction from [6]:

▶ **Definition 4** (oracle)**.** When a client accesses an unavailable configuration C it gets an error message referencing some activated configuration $C' \not\subseteq C$.

3 Reconfiguration Problem

We want to allow clients that access a shared object to change the subset of servers over which it is emulated. To this end, we define a reconfiguration abstraction, which has one operation, *reconfig*. A *reconfig* operation gets as parameters a configuration C and a proposal $P \subset Changes$. Intuitively, *reconfig*(C, P) is a request to reconfigure the system from configuration C to a new configuration reflecting the changes in P. It returns two values. The first is a configuration C' which is either $C \cup P$ or a superset of it, i.e., $C' \supseteq C \cup P$, where C' may contain additional, concurrently proposed changes. It also returns a set S consisting of all the configurations that were speculated during the operation, and in particular, $C' \in S$. We assume that C is a configuration that was previously returned by some *reconfig* operation (note that this is an assumption on usage). By convention, we say that *reconfig*(C_0, C_0) returns $\langle C_0, \{C_0\}\rangle$ at time 0.

Next we need to determine when configurations are activated. Since one of the purposes of reconfiguration is to allow administrators to switch off removed servers, we want to make sure that *reconfig* leads to the activation of a new configuration, which in turn expires old ones. However, since at the moment when a configuration is activated all preceding configurations may become unavailable, we want to allow clients to transfer object state residing in the old configuration to the new one. Clearly every distributed service maintains *some state* on behalf of clients. Thus, when reconfiguring, we need to be careful to not lose this state. We want to define a generic way, without any specific knowledge of the higher

[1] In today's practical settings, it is reasonable to presume that some global directory is available, e.g., DNS.

level service, to make clients aware of a reconfiguration that is about to happen so they will be able to transfer state to it. Therefore, in our model, a configuration C is not necessarily immediately activated when *reconfig* returns it. Instead, when *reconfig* returns C, our model allows clients to transfer state from previous configurations to C. Only when a client calls *reconfig*(C, P) (for some P), and returns C (indicating no further changes) does C become activated. Formally:

▶ **Definition 5** (activation)**.** A configuration C is *activated* when *reconfig*(C, P) returns $\langle C, S\rangle$ for some S and P for the first time.

Note that by the convention, the initial configuration C_0 is activated at time 0.

Sequential specification The reconfiguration abstraction is linearizable with respect to the sequential specification consisting of the three properties we now define. Briefly, the idea is that we require that *reconfig* return to all clients configurations that are totally ordered by containment. Importantly, we do not require *reconfig* to return to every client the entire totally-ordered sequence. Rather, we allow clients to "skip" configurations.

First, we require that every change in a speculated configuration was previously proposed:

- **Validity:** A *reconfig* operation *rec* returns $\langle C, S\rangle$ s.t. for every $C' \in S$ for every $e \in C'$, $\exists$*reconfig*(C'', P') that is either *rec* or precedes it s.t. $e \in P'$.

Second, we require that new configurations contain all previous ones:

- **Monotonicity:** If $\langle C, S\rangle$ is returned before $\langle C', S'\rangle$, then $C \subseteq C'$.

Note that it is possible for one client to activate C' after C, while another client reconfigures C to another configuration C''. The third property uses speculation to ensure that the second client is aware of configuration activated by the first:

- **Speculation:** If a configuration $C' \supset C$ is activated before *reconfig*(C, P) returns $\langle C'', S\rangle$, then $C' \in S$.

Liveness In addition, if the number of invoked *reconfig* operations is bounded, we require:

- **Termination:** Every reconfig operation invoked by a correct client eventually returns.

It is easy to see that in this model (servers and clients are separated), if there is an unbounded number of reconfigurations' invocations, then a correct client may forever chase after an available configuration and never be able to communicate with the servers, and thus, never complete its operation[2].

4 Reconfiguration Solution

In this section we give a generic algorithm for the reconfiguration problem. In Section 4.1 we define the Speculating Snapshot (SpSn) abstraction, which is the core task behind the algorithm. In Section 4.2 we use the SpSn abstraction in order to present a generic reconfiguration algorithm. In Section 4.3 we show different ways to implement SpSn by recasting existing algorithms of atomic dynamic storage in terms of SpSn.

[2] In [11], we show that even in a model where clients are not distinct from servers, and only clients that are part of the last activated configuration's membership are allowed to invoke operations, we cannot guarantee progress in case of infinite number of reconfiguration even if we can solve consensus in every configuration.

Table 1 Possible SpSn outputs.

Client	Input	Output
c_1	$\{+s_1\}$	$\{\ \{\{+s_1\}\},\ \{\{+s_1\},\{+s_2\},\{+s_3\}\}\ \}$
c_2	$\{+s_2\}$	$\{\ \{\{+s_1\}\},\ \{\{+s_1\},\{+s_3\}\}\ \}$
c_3	$\{+s_3\}$	$\{\ \{\{+s_1\}\},\ \{\{+s_1\},\{+s_2\}\},\ \{\{+s_1\},\{+s_2\},\{+s_3\}\}\ \}$
c_4	$\{\}$	$\{\}$

4.1 SpSn abstraction

SpSn (based on [6]) is the core task clients solve in configurations in order to coordinate configuration changes. It is a multi-input, multi-output task: each client inputs its proposal P by calling *SpSn(P)*, and the output is a set of sets of proposals proposed by different clients. We will use SpSn for *reconfig* by proposing changes, and each of the sets returned by SpSn will be speculated. SpSn is emulated in a given configuration C, and its invocation in C with proposal P is denoted $C.SpSn(P)$. Like other emulated objects, C.SpSn can return an error message with some newer activated configuration if C is unavailable. Within an available configuration C, the SpSn task is defined as follows:

- **Non-triviality:** If $P \not\subseteq C$, then *SpSn(P)* returns a non-empty set.
- **Intersection:** There exists a non-empty set of proposals that appears in all non-empty outputs.

An example of possible SpSn outputs appears in Table 1.

4.2 Generic algorithm for reconfiguration

In this section we show a simple generic algorithm for the reconfiguration problem, which is based on the dynamic storage algorithm presented in DynaStore [1]. We use one SpSn task in every configuration. When clients call $C.SpSn$ with different proposals, they may receive different configurations in return, which in turn leads them to speculate different configurations.

The pseudocode of the algorithm appears in Algorithm 1. The idea is to track the configurations that clients speculate, and try to merge them into one configuration that will reflect them all. In addition, we want to make sure that later *reconfig* operations will be aware of this configuration in order to guarantee monotonicity. We call this process *traverse* [1] because it is a traversal of a configuration DAG (see Figure 2 above) whose nodes are the speculated configurations and there is an edge from a configuration C to a configuration C' if some client receives C' in the output of $C.SpSn$.

Initially, the set *ToTrack* contains only the input configuration C in which the operation started, *proposal* is the union of C and the input proposal P, and the set *speculation* is empty. During the reconfig operation, a client repeatedly takes the smallest configuration in *ToTrack*, speculates it, adds it to the *speculation* set, and proposes *proposal* in its SpSn. Then, if the configuration is available, it adds the output from SpSn (set of configurations) to *ToTrack* (in order to track them later), and adds the union of all the changes in configurations returned from SpSn to *proposal*. Note that each client traverses a different sub-graph of the DAG of configurations during its *reconfig* operation. However, since SpSn guarantees intersection, the DAGs of different clients that start in the same configuration intersect (see example in Figure 3). A *reconfig* operation completes when *ToTrack* is empty. The last configuration in *ToTrack* is the configuration where the DAGs merge.

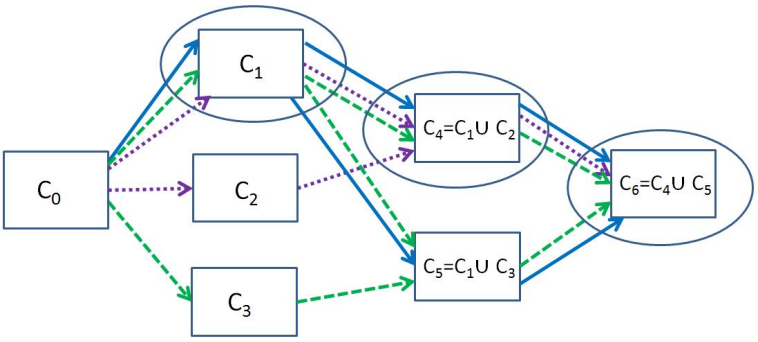

Figure 3 Clients c_a, c_b, and c_c start *reconfig* in configuration C_0. Client c_a traverses the solid blue arrows, c_b traverses green dashed arrows, and c_c traverses purple dotted arrows. For example, c_a first invokes $C_0.SpSn(C_1)$ and receives $\{C_1\}$. Next it invokes $C_1.SpSn(C_1)$ and receives $\{C_4, C_5\}$. Then, it invokes $C_4.SpSn(C_6)$ and $C_5.SpSn(C_6)$ and receives $\{C_6\}$ from both them. Finally, it invokes $C_6.SpSn(C_6)$, receives $\{C_6\}$, and returns (no more configurations to track) C_6 together with a set consisting of the configurations in its DAG. Each of the clients traverses a different sub-graph but since SpSn guarantees intersection, their traversals intersect and eventually merge. The circled configurations are those where the sub-graphs intersect.

While traversing a DAG, if some configuration is unavailable, a client receives an error message with a newer activated configuration C_a, and starts over from C_a. Note that since we assume a bounded number of reconfigurations, clients with pending operations will (1) eventually reach an available (forever) configuration, and (2) propose the same configuration. Therefore, all the operations eventually complete.

Algorithm 1 Generic algorithm for reconfiguration

```
1:  operation reconfig(C, P)
2:      ToTrack ← {C}
3:      proposal ← P ∪ C
4:      speculation ← {}                                  ▷ set of speculated configurations
5:      while ToTrack ≠ {} do
6:          C' ← argmin_{C''∈ToTrack} (|C''|)             ▷ smallest configuration (in number of changes)
7:          speculate(C')
8:          speculation ← speculation ∪ {C'}
9:          ret ← C'.SpSn(proposal)
10:         if ret = ⟨"error", C_a⟩ then                  ▷ C' is expired - restart from C_a
11:             speculation ← {}
12:             ToTrack ← {C_a}
13:         else
14:             ToTrack ← (ToTrack ∪ {⋃_{e∈E} e | E ∈ ret}) \ {C'}
15:         proposal ← ⋃_{e∈ToTrack} e ∪ proposal
16:     return ⟨proposal, speculation⟩
17: end
```

Table 2 Comparison among SpSn implementations extracted from existing dynamic storage algorithms

Algorithm	SpSn Cost	DAG size	reconfigurable	rely on consensus
RAMBO [7, 8]	O(1)	n	yes	yes
DynaStore [1]	O(1)	$min(mn, 2^n)$	yes	no
SmartMerge [9]	O(1)	n	no	no
Parsimonious SpSn [6]	O(n)	n	yes	no

4.3 Recasting existing algorithms in terms of SpSn

In this section we look into existing algorithms and extract their core mechanism for implementing SpSn. As noted above, we assume a shared memory abstraction in every configuration, and as long as the configuration is available, clients can access its read/write registers. Therefore, we implement SpSn in shared memory, and when a configuration C becomes unavailable, pending SpSn invocations in C return an error message with some active configuration.

We make use of a *collect* operation, which returns a set of values of an array of registers. This operation can be implemented by reading the registers one by one, or by opening the ABD abstraction and collecting an entire array of registers in a constant number of communication rounds. In our complexity analysis, we count a collect as one operation.

The SpSn implementations differ in their complexity (number of operations) and in the number of configurations clients traverse in the generic reconfiguration algorithm (i.e., their DAG size). Denote by m the total number of *reconfig* operations, where n of them propose unique changes. As we will see in Section 5, when emulating a dynamic atomic register, read and write operations invoke *reconfig* without proposing changes (they call *reconfig* in order to ensure they execute in the up-to-date configuration), while reconfigurations of the register propose changes. Table 2 compares the different SpSn implementations described in detail below.

4.3.1 RAMBO

RAMBO [8, 7] was the first to implement a dynamic atomic register with asynchronous read/write operations. The main idea is to use consensus to agree on the reconfigurations, while *read/write* operations asynchronously read from all available configurations and write to the "last" one. We now show how to use consensus in order to implement SpSn. The pseudocode appears in Algorithm 2. It uses a shared array arr where client c_i writes to $arr[i]$. We assume that arr is dynamic: only cells that are written to are allocated.

Consider a client c_i that proposes P in SpSn of configuration C. If $P \not\subseteq C$ (meaning that the client has new changes to propose), it proposes P in C's consensus object, and writes the decision value to its place in arr. Otherwise, it does not invoke consensus and writes nothing to arr. In both cases, it returns the set of sets of values collected from arr. (Only written cells are collected). Note that this set is either empty, in case no changes were proposed, or contains exactly one set of one configuration (the one agreed in the consensus). Therefore, the SpSn non-triviality and intersection properties are preserved.

Note also that clients invoke consensus only if they propose new configurations. Thus, if we use this SpSn in the register emulation of Section 5.2, we preserve the RAMBO property of asynchronous read/write operations.

Algorithm 2 Consensus-based SpSn; protocol of client c_i in configuration C

```
1: operation SpSn(P)
2:     if P ⊈ C then
3:         arr[i] ← C.consensus(P)
4:     ret ← collect(arr)
5:     return {{C'} | C' ∈ ret}
6: end
```

4.3.2 DynaStore

DynaStore [1] was the first algorithm to solve dynamic storage reconfiguration in completely asynchronous systems (without consensus). It observes that clients do not have to agree on the next configuration: different clients can return different configurations, as long as we make sure that if one client writes in some configuration, others will traverse this configuration, read its value, and transfer it to the new configuration they return.

The core mechanism behind the coordination of the algorithm is the weak snapshot abstraction. We now show how to use it in order to implement SpSn. The pseudocode of client c_i implementing *SpSn(P)* in configuration C appears in Algorithm 3. Again, we use an array arr. If c_i proposes a new configuration ($P \not\subseteq C$), then it writes P into its register in arr. Otherwise, it writes nothing. Then it collects the registers in arr. If the collect is empty, it returns {}, otherwise it collects again and returns the obtained set.

Note that both properties of SpSn are preserved. First, non-triviality is satisfied since if $P \not\subseteq C$ then the client writes to its register and so the collects cannot return an empty set. Second, intersection is satisfied since all the clients that return non-empty sets get a non-empty set in the first collect, and thus collect again. Therefore, in the second collect they all get the first value that is written (this value appears in all outputs).

Note that while the DAG size obtained in the generic algorithm by using consensus for SpSn is exactly n, with a weak snapshot-based SpSn, the DAG can be much bigger. Without consensus, clients can write (propose) different configurations in SpSn's array (arr), and learn different subsets of proposals (from the *collect*), which in turn leads to different proposals being written in the next tracked configuration's SpSn.

With n is unique proposals, there are 2^n possible configurations that can be speculated and traversed. But since clients propose in each SpSn during the traverse the union of all the configurations and proposals they previously traversed, every client proposes at most n different configurations during its traverse. Therefore, the worst-case DAG size is $min(nm, 2^n)$.

Algorithm 3 Weak snapshot-based SpSn; protocol of client c_i in configuration C

```
1: operation SpSn(P)
2:     if P ⊈ C then
3:         arr[i] ← P
4:     ret ← collect(arr)
5:     if ret = {} then
6:         return ret
7:     else
8:         ret ← collect(arr)
9:         return {{C'} | C' ∈ ret}
10: end
```

4.3.3 SmartMerge

SmartMerge [9] is very similar to DynaStore, but it uses a pre-computation in order to reduce the DAG size to n. Before starting the generic algorithm, clients participate in an external lattice agreement service [4], in which they input their proposals and each receives a set of proposals s.t. all the outputs are related by containment. Then, they take the output of the lattice agreement and use it as their proposal in the generic *reconfig* algorithm (Algorithm 1).

Notice that by ordering the proposals by containment, SmartMerge reduces the total number of configurations that can be speculated and traversed (i.e., the DAG size) to n. However, this solution assumes that the lattice agreement service is available forever, and since it is not a dynamic service, the servers emulating it cannot fail or be switched off. Therefore, SmartMerge is not reconfigurable.

4.3.4 Parsimonious SpSn

Parsimonious SpSn [6] uses multiple rounds of a mechanism similar to commit-adopt [5]. Similarly to SmartMerge, it relies on containment in order to reduce the DAG size, but does not use an external service for it, and thus the solution is reconfigurable. Instead, all the configurations in the sets returned from the commit-adopt-based SpSn are related by containment.

In order to achieve the containment property, parsimonious SpSn pays in the SpSn's complexity. Instead of O(1) as in other implementations, the SpSn complexity here is O(n). More details can be found in [6].

5 Dynamic Atomic Register

The reconfiguration problem can be used as an abstraction in order to implement many dynamic atomic objects on top of it. Here we demonstrate it by presenting a protocol for dynamic atomic register. In Section 5.1 we define the dynamic atomic register object, and in Section 5.2 we present an algorithm that implements it in our model.

5.1 Definition

We consider a dynamic atomic multi-writer, multi-reader (MWMR) register object emulated by a subset of the servers in ϕ, from which any client can *read* or *write* values from a domain $\mathbb{V}$. The sequential specification of the register requires that a *read* operation return the value written by the latest preceding *write* operation, or $\perp$ if there is no such write. In addition,

the object exposes an interface for invoking *reconfiguration* operations that allow clients to change the set of servers emulating the register.

A *reconfiguration* gets as a parameter a set of changes $Proposal \subset Changes$ and returns a configuration C s.t. (1) C is activated, (2) $C \supseteq Proposal$, and (3) C is subset of changes proposed by clients before the operation returns.

We assume that there is a bounded number of *reconfiguration* operations, and require that every operation by a correct client eventually returns.

5.2 Solution

We present an algorithm for a dynamic atomic register, which uses the reconfiguration problem abstraction (*reconfig*). The pseudocode appears in Algorithm 4.

The main procedure used by all operations, (*read*, *write*, and *reconfiguration*), is *check-config*(P, v, op), where P is the reconfiguration proposal ($\perp$ in case of *read* or *write*), $v \in \mathbb{V}$ is the value to write ($\perp$ in case of *read* or *reconfig*), and *op* is the operation type (READ, WRITE, or REC). The procedure manipulates two local variables: C_{cur}, which stores the last activated configuration returned from a *reconfig* operation, and *version*, a tuple consisting of a value v and its timestamp ts. All operations first call *check-config*(P, v, op), and then return according to the operation type: A *write* returns ok, a *read* returns $version.v$, and a *reconfiguration* returns C_{cur}.

In order to emulate the dynamic register we use an idea that was first introduced in RAMBO [7], and later adopted by DynaStore [1]. The idea is to read a version from each configuration that other clients may have written to, and then write the most up-to-date version (associated with the highest timestamp) to the configuration we want to activate and return. To this end, we start *check-config* by calling *reconfig*$(C_{cur}, Proposal)$, which returns $\langle C, S \rangle$. Next, we read the version from every configuration returned in S, and write the latest/newer version into C. In case of a *write* operation we write a version consisting of v and a new timestamp (higher than all those we read). Otherwise, we write back the version with the highest timestamp we read.

Note that before we can return, we need to validate that future operations will not miss our version. Therefore, after we write the version, we check if there are new activated configurations. To this end, we call *reconfig*$(C, \{\})$ (line 27). If the operation returns $\langle C', S' \rangle$ where $C' = C$, it is guaranteed that no one "moved forward" before we wrote our version to C, and every later operation will not miss our version. Note that in this case, by our definition, C is activated and older configurations can become unavailable. This does not pose a problem, since the state of the object (the up-to-date version) has already been transferred to the new configuration. Otherwise, if the operation returns $\langle C', S' \rangle$ where $C' \neq C$, we repeat the above process for C' and S'. Notice that since we assume a bounded number of *reconfiguration* operations, it is guaranteed that every *check-config*, and thus every operation performed by a correct client eventually returns.

In order to read and write versions from configurations we assume that every configuration emulates a version object that has two functions: *readVersion*() and *writeVersion*$(version)$. A *readVersion* invoked in configuration C simply returns C's version. A *writeVersion*$(version)$ overwrites C's version if it has a higher timestamp, and returns ok. Again, if C is unavailable, the operations return error messages. Note that *readVersion*() can be implemented by the first phase of ABD [2], and *writeVersion* by the second.

Algorithm 4 Dynamic Atomic register emulation

```
1:  Local variable:
2:      version, tmp ∈ ℕ × 𝕍 with selectors ts and v, initially ⟨0, v_0⟩
3:      C_cur ⊂ Changes, initially C_0

4:  operation reconfiguration(Proposal)
5:      check-config(Proposal, ⊥, REC)
6:      return C_cur
7:  end

8:  operation read()
9:      check-config(⊥, ⊥, READ)
10:     return version.v
11: end

12: operation write(v)
13:     check-config(⊥, v, WRITE)
14:     return ok
15: end

16: procedure check-config(P, v, op)
17:     ⟨C, S⟩ ← reconfig(C_cur, P)
18:     repeat
19:         for each configuration C′ ∈ S do          ▷ read in all speculated configurations
20:             tmp ← C′.readVersion()
21:             if tmp ≠ error(∗) ∧ tmp.ts > version.ts then          ▷ newer version found
22:                 version ← tmp
23:         if op = WRITE then
24:             version ← ⟨version.ts + 1, v⟩
25:         C.writeVersion(version)
26:         C_tmp ← C
27:         ⟨C, S⟩ ← reconfig(C_tmp, {})          ▷ activate C or find newer configuration
28:     until C = C_tmp
29:     C_cur ← C                                          ▷ C is activated
```

6 Conclusion

Reconfiguration is a key challenge in implementing distributed dynamic shared objects, and in particular, in distributed dynamic storage. Clearly, any long-lived shared object emulated on top of fault-prone servers must be able to reconfigure in order to remove failed or old servers and add healthy or new ones.

In this tutorial we first defined a clear model for studying reconfiguration. We defined a failure condition that provides reconfigurability, that is, allows a server to fail or be switched off immediately when it is no longer part of the current active configuration's membership. Then, we encapsulated a reconfiguration problem that is on the one hand implementable in asynchronous systems satisfying our failure condition, and on the other hand can be used as an abstraction for implementing many dynamic shared objects. Next, we presented a (simple)

general framework for solving the reconfiguration problem, and showed how existing dynamic storage algorithms [7, 8, 1, 9, 6, 10] can be recast in terms of this framework. In order to do so, we defined, (based on [6]), a core task called SpSn, which clients solve in order to coordinate. We demonstrated how to extract different algorithms' coordination mechanisms in order to implement this task. This allowed us to compare the different algorithms.

Finally, we demonstrated the power of the reconfiguration abstraction by presenting a simple algorithm for dynamic atomic storage on top of it.

Acknowledgements

We thank Eli Gafni for his contribution to the SpSn abstraction definition, and Christian Cachin and Yoram Moses for insightful comments.

References

1. 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.
2. 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.
3. Gregory Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: A comprehensive study. *ACM Computing Surveys (CSUR)*, 33(4):1–43, December 2001.
4. Jose M Faleiro, Sriram Rajamani, Kaushik Rajan, G Ramalingam, and Kapil Vaswani. Generalized lattice agreement. In *Proceedings of the 2012 ACM symposium on Principles of distributed computing*, pages 125–134. ACM, 2012.
5. Eli Gafni. Round-by-round fault detectors (extended abstract): unifying synchrony and asynchrony. In *Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*, pages 143–152. ACM, 1998.
6. 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.
7. 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.
8. 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.
9. 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.
10. 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.
11. Alexander Spiegelman and Idit Keidar. On liveness of dynamic storage. *CoRR*, abs/1507.07086, 2015.