Elastic Configuration Maintenance via a Parsimonious Speculating Snapshot Solution

Eli Gafni[1] and Dahlia Malkhi[2]

[1] UCLA, eli@cs.ucla.edu
[2] VMware Research, dmalkhi@vmware.com

1 Introduction

In order to provide dynamic reconfiguration of a distributed service, we extract a fundamental new task SpSn. This new task facilitates a consensus-free coordination among *clients* on incorporating changes to the set of *servers* they all access, and through which they negotiate the changes. The danger is of course that when transitioning from one configuration to the other, the system might break up isolating the clients into several groups that cannot communicate with each other.

SpSn. We start with a formal definition of the new task in a generic from. A processor p_i invokes the task with input I_i, and returns a pair (Q_i, Y_i), where for some contextual value-space U, $I_i, Q_i \subseteq U$, and $Y_i \subseteq 2^U$, and such that:

1. $Q_i \subseteq \cup_{j \in players} I_j$, where $players \subseteq clients$ is the set of participating clients, $I_i \subseteq Q_i$, and the Q_j's returned are related by containment, and
2. For all i, j if $Q_j \subseteq Q_i$, then $Q_j \in Y_i$.

Since the outputs are snapshots of the inputs, as well as a "speculation" of any output that earlier processors might have obtained, we name the task *Speculating Snapshot*, in short SpSn.

In our context the input for the task per client is a *configuration change* proposal from a set $P = \{+s, -s\}_{s \in servers}$[3].

Parsimonious Solution to SpSn. A possible solution to SpSn is for Q to be a snapshot of the inputs, and for Y to be the power-set of Q, i.e. the set of all subset of Q. However, this is inefficient in the number of configurations which clients observe in our use of SpSn to affect a configuration change. We later show why existing solutions to the dynamic configuration problem contain a solution to SpSn, and how the complexity of a reconfiguration scheme is related to the various solutions to SpSn. Here, we will be concerned with the most parsimonious solution in terms of the cardinality of Y_i. If we solve SpSn by consensus on a

[3] We assume that each server is added and removed at most once, so to be re-introduced into the system it bears a new identity.

total order of configuration-changes, we can get Y_i to be precisely the number of previously output snapshots. The number of configurations here is linear in the number of proposals. However, the worst-case cost here is an infinite execution, as mandated by the FLP impossibility theorem [5].

We want to be as parsimonious as consensus-based solutions without relying on consensus. Briefly, a parsimonious solution to SpSn wait-free SWMR Read-Write is to go through a sequence of phases. Each phase is built around a two-step protocol which posts a proposal and collects all other proposals. In the second step, if the first collect was uniform, a processor marks it as a *commit proposal*. This structure borrows from the Commit-Adopt building block of Gafni [6]. At the end of the second step, if a commit value is unanimous, processor p_i returns it as Q_i. Otherwise, it accumulates all commit values in Y_i and continues to another phase. The body of the paper contains a precise description of this solution.

Wheras so far, we expressed the solution in a shared-memory model, it may be implemented distributed and fault tolerant using the Read-Write register emulation due to Attiya et al. [3]. In the body of the paper, we first describe our solution to SpSn using shared registers for abstraction (Section 3). This requires pre-allocating registers per client, hence the solution is not adaptive. We then "open" the shared register emulation and derive an *adaptive* solution (Section 4).

The Dynamic Reconfiguration problem. We now discuss how we use the above solution SpSn in solving the *Dynamic Reconfiguration* problem. In this problem there is an initial configuration of *servers* known to all clients. We say that clients are initially *subscribed* to this initial configuration. In a one-shot Dynamic Reconfiguration problem, every client process in a subset $players \subseteq clients$ proposes one configuration-change from P. (We will later discuss the long-lived Dynamic Reconfiguration problem.) The goal is for all *players* to eventually subscribe to a common, final configuration encompassing all the proposals. Recall that a set of proposals uniquely defines a set of *servers*, hence we focus on converging on the set of proposals.

It should be understood that although there is no a priori bound on the number of steps taken until convergence, any solution must, at some point, allow a client to subscribe to a new configuration. However, this may not necessarily be a final subscription, because the set *players* is not a priori known, and proposals may continue arriving. Hence, even after it subscribes to a new configuration, a client must continue observing other proposals written into its latest subscribed configuration, and potentially subscribe to a newer configurations. Our problem definition mandates that after all proposals have arrived, all *players*, provided they take enough steps, will subscribe to a final configuration that will not change. The clients themselves may not know that this is the final configuration (namely, the set *players* of participating clients is unknown to clients).

We now return to discuss long-lived Dynamic Reconfiguration. In this problem, clients, over time, are not restricted to request just one change. We can reduce Dynamic Reconfiguration into the problem of one shot by conceptualizing a new change request by a client as a new client. Obviously, that new virtual client can start at the configuration that the old virtual client ended.

If clients were able to access the initial configuration forever, the Dynamic Reconfiguration problem would not be hard to solve. Clients would simply repeatedly collect proposals written to the initial configuration and output their union. In our problem, once a client subscribes to some configuration S, it 'expires' previous configurations in the following sense: Every configuration S', where $S' \subseteq S$, may stop receiving new change proposals. (How they might "know" that they expired is of no concern here.) This is what makes the problem useful: Clients better not diverge into disjoint configurations, because there would be no way for them to find out about each other and converge back.

Having stated the problem, we can easily see how to solve it using SpSn. Every client participates in implementing SpSn using the set of servers in the initial configuration. The client provides its input proposal to SpSn. While solving SpSn in the initial configuration it accumulates sets into its Y. These sets might have been subscribed to by other processors. Hence it now solves SpSn in each one of them. There are two ways to do it, one akin to depth first, and one to breadth first. We comment on the latter in the conclusions. In the former, the client solves SpSn in each of the speculated configurations in its Y set, one by one. The output from one SpSn is the input to the next. It does this until its input to SpSn is the same as the output from it. Only then it subscribes to that configuration.

Garbage Collection. We intentionally formulated our dynamic problem without modeling failures, focusing only on the necessary ingredients to guarantee that information can be passed from one configuration to the next. As a practical matter, it is worth noting that since we expire old configurations, we may garbage collect their resources and not have to rely on their availability. What will a client do if it cannot perform SpSn in a configuration S which it is subscribed to? We stipulate that the client can be notified by an auxiliary mechanism that a new configuration subscription caused S to expire. Note that this is a very weak assumption, it only provides a client with eventual expiration notification on a configuration S which it already subscribed to. Relying on an external 'oracle' notifications after old configurations are garbage collected is inherent in dynamic systems, see e.g., [9, 2].

Application. We demonstrate a use-case of consensus-free Dynamic Reconfiguration, a dynamically reconfigurable store. A single-register store is built by interjecting reads and writes of the register during Dynamic Reconfiguration, and likewise observing configuration information during normal Read/Write operation. During reconfiguration, a client reads the register within every configuration in Y and writes it into to the output configuration S. Within a Write operation, a client starts at the latest subscribed configuration and performs write–then–SpSn. It repeats this for every speculated configuration in Y. A Read starts with the latest subscribed configuration, calls SpSn and then iterates within every configuration in Y doing SpSn–then-read. Correctness intuitively stems from the fact that in every configuration in which a Write is performed, either a Read observes the value, or the written value is first copied to a newer config-

uration. Section 6 contains a brief description of dynamic store algorithm and a correctness sketch.

Organization. The remainder of the paper is organized as follows. We comment on related works in Section 1.1. A formal execution model is provided in Section 2. The solution is laid out in two parts. First, in Section 3 we solve SpSn Read-Write. Then we provide a distributed SpSn protocol in Section 4. We use SpSn as building block to solve the Dynamic Reconfiguration problem in Section 5. We briefly outline the design of an elastic Read-Write store utilizing SpSn in Section 6. We conclude and discuss future work in Section 7.

1.1 Related work

Much of our modularity owes to two prior celebrated results, the Read-Write register emulation of Attiyah, Bar-Noy and Dolev (ABD) [3], and the Commit-Adopt protocol of Gafni et al. [6].

Our story begins with the ABD emulation which provides an atomic Read-Write service over a fixed collection of $2F + 1$ servers, F of which may become unavailable. In a nutshell, the emulation is built of two communication phases. One is used for querying about the currently stored value and its timestamp. The second one is used for updating the stored value and its timestamp. Each phase employs a majority-exchange, guaranteeing intersection with past phases in at least one server. Our SpSn emulation is first presented using Read-Write registers within each configuration as building block for modularity. We then leverage the ABD fault-tolerant emulation within each fixed configuration to derive a message-passing protocol.

To make the ABD emulation *elastic*, the pioneering work of RAMBO integrated a configuration consensus service to facilitate reconfiguration; the first RAMBO works operated the configuration service separately from the emulation [9, 7], and later, it became intertwined with the register emulation itself [4]. That work opened the formal treatment and definition of elastic problems.

SpSn identifies the crux of a reconfiguration task which is embedded in such elastic solutions. Indeed, a degenerate form of SpSn occurs in any dynamic system which employs consensus for configuration such as RAMBO [9]. Here, every client is handed a global sequence of configurations. Each prefix of the sequence could be reduced to an SpSn output by "speculating" every prefix of the sequence. Consensus-based reconfiguration is parsimonious in the number of configuration-changes, but relies on the strength of consensus. Our interest is in consensus-less elasticity.

The most relevant prior work is DynaStore [2, 11], a previously known consensus-free dynamic store. The complexity of DynaStore's SpSn is an exponential number of configurations. This can be easily seen as follows. A client in DynaStore starts with the last known configuration and participates in implementing with the servers in this configuration a new primitive named *Weak Snapshot*. Weak Snapshot returns to every client a collection of proposals, with one common proposal included in all collections, which are otherwise otherwise unconstrained.

With n proposals, there are $2^{(n-1)}$ possible such collections, which DynaStore clients traverse in order to converge on a final configuration. There are other differences between SpSn and DynaStore along several dimensions.

- Relying on the strong foundations of atomic Read-Write registers and Commit-Adopt, we provide a fairly succinct and modular solution, which is described in less than 20 LOC. Although elegance is an elusive property, we feel that a deductive re-visit is warranted given the importance of the DynaStore contribution.
- DynaStore provides reconfigurable atomic Read-Write storage. We provide a modular approach which separates reconfiguration as a building block by itself.

More generally, dynamic storage is a fundamental service which received tremendous attention in both theory and practice, beyond the scope we can cover here. We refer the reader to two recent surveys which may shed light into this arena: A tutorial on foundations is given in [1], and a more broad survey which covers both theory and practice is provided in [10].

2 Problem model

The introduction already introduces the participants: A set *clients* of client-processes and a set *servers* of server-processes, and a subset $players \subseteq clients$ participating in solving the SpSn tasks and in Dynamic Reconfiguration. We proceed to formally indicate the execution paradigm and the interaction model among participants.

We consider two coordination models. In one, processes use shared atomic single-writer multi-reader (SWMR) Read-Write registers. Each register r provides two operations, r.read and r.write. Each process may invoke one operation on any register and wait for it to return. There is no a priori bound on operation execution times nor on processing speeds of processes. That is, the system is asynchronous. An execution may interleave operations by different clients on registers. For every execution, there exists an equivalent sequential execution in which read and write operations return the same results as the real execution, and furthermore, the sequential execution respects real-time ordering between non-overlapping operations in the real execution. That is, every execution is *linearizable*. For a formal treatment of atomic registers, executions, execution equivalence and linearizability, we refer the reader to the classic literature [12].

Our second coordination model uses messages for communication between client-processes and server-processes. There is no a priori bound on message transfer times between clients, but it is guaranteed that message origins are authentic and that messages between live processes arrive in tact. That is, the system employs the standard asynchronous message-passing model [12].

Configurations. A *configuration* is expressed in one of two interchangeable forms. One is simply as a subset $S \subseteq servers$. The other is as a *change-set* $S \subseteq \{+s, -s\}_{s \in servers}$. The latter form reduces to a subset by subtracting all the servers s included in $-s$ form, from those in $+s$ form.

Availability and Garbage Collection. In order to capture system elasticity, we model configurations as being either *Active* or *Expired*. In the shared-memory model, an Active configuration provides clients with access to atomic Read-Write registers belonging to the configuration. When a client tries to write a register of an expired configuration, the environment throws an exception indicating that the configuration is expired, and provides the cause of its expiration. In the message-passing model, an Active configuration has a majority of servers available and responsive to client messages. An Expired configuration in the message-passing model is the same as a shared-memory one, and notifies clients that attempt to access it about its expiration through an exception.

The set of Active configurations is determined as follows. Initially, the system starts with an a priori fixed Active configuration C_0. Whenever SpSn returns Y to some client, every configuration $C \in Y$ becomes Active.

At any moment, every participating client has a single configuration which it is *subscribed* to. Clients start by default subscribed to C_0. During an execution, a client may adopt a configuration and subscribe to it. This may occur an unbounded number of times. If a client crashes, we proforma regard it as if the client remains subscribed to the last configuration is was subscribed to before the crash; this has no effect on our problem specification or solution.

To allow garbage collection, when a client subscribes to a configuration S, the subscription to S causes every configuration S' such that $S' \subseteq S$ to become *Expired*.

3 SpSn Read-Write Solution

This section provides a solution for the SpSn problem defined in the Introduction. The procedure C.SpSn(I_i) in Algorithm 1 captures the actions of a client-process p_i whose input is I_i.

The solution builds around a two-*st*ep protocol which is repeatedly invoked. The protocol bears similarity to the Commit-Adopt procedure of Gafni [6]. For each client p_i and for each internal *phase* counter $k = 1, 2, ...$, the implementation uses two SWMR shared atomic registers, $I_i(k, 1)$ and $I_i(k, 2)$. In phase k, at the first step, a client p_i first writes its proposal to $I_i(k, 1)$ and then collects all written $I_j(k, 1)$ values. If all the values it observes are identical, in the second step it writes to $I_i(k, 2)$ a *commit-proposal* (with a commit-bit set) with this value. Otherwise, it proposes as non-commit a union of all values it observed in the first step. It then collects all written $I_j(k, 2)$ values. Every commit-proposal is kept in Y_i.

If any non-commit proposal $I_j(k, 2)$ was collected, then another phase is started. In the next phase, the initial proposal is the union of all the $I_j(k, 2)$

values. Otherwise, if all the $I_j(k,2)$ values are commit proposals (a fortiori, they are all identical), then SpSn returns this value as Q_i, along with the set Y_i. At this point, Y_i contains all commit values which were accumulated in preceding phases.

We remind that the formulation of SpSn in shared-memory is for pedagogical purposes. Section 4 gives a message passing implementation which is also *adaptive* and does not require prior knowledge of the client-set.

Algorithm 1 *C.SpSn* protocol at process p_i

```
1: local variables:
2:     proposal, collect, commit, w
3:     Y_i, initially ∅
4:
5: procedure C.SpSn(I_i)
6:
7:     proposal ← I_i
8:     for k = 1, 2, 3, ... do
9:                                                          ▷ first phase
10:        commit ← true, collect ← ∅
11:        I_i(k, 1).write(proposal)
12:        for every client p_j do
13:            w ← I_j(k, 1).read
14:            if w ≠ ∅ and w ≠ proposal then
15:                collect ← collect ∪ w, commit ← false
16:                                                         ▷ second phase
17:        I_i(k, 2).write(⟨commit, collect⟩)
18:        for every client p_j do
19:            ⟨w.commit, w.set⟩ ← I_j(k, 2).read
20:            if w.commit ≠ true then
21:                commit ← false
22:            if w.commit == true then
23:                Y_i ← Y_i ∪ w.set
24:            proposal ← proposal ∪ w.set
25:
26:        if commit == true then
27:            return proposal, Y_i
```

Correctness of SpSn RW Solution

Lemma 1. *In each phase k of the SpSn procedure, if any two commit values are written to $I_i(k,2)$, $I_j(k,2)$ (i.e., both have the commit bit set), then they are the same. Furthermore, the value must be the first value whose write in the first step of phase k has completed.*

Proof. Fix some k, and let p_f be the client whose write into $I_f(k,1)$ is the first to complete. Let p_i be any client writing a commit value to $I_i(k,2)$. Therefore, the

collect of all $I_j(k,1)$ by p_i returned identical values. Furthermore, by assumption, p_i's read of $I_f(k,1)$ must have returned the value written by p_f. Therefore, p_i's unanimous collect value must be $I_f(k,1)$, and the lemma is proved.

Lemma 2. *Procedure SpSn in Algorithm 1 maintains the properties listed under the SpSn problem definition in the Introduction.*

Proof. Property 1 of SpSn in the Introduction has two components, Validity and Containment. The Validity property that $I_i \subseteq Q_i$ immediately follows from the fact that a process p_i first writes its own proposal into $I_i(k,1)$ and then collects all $I_j(k,1)$.

To prove Containment, note that by Lemma 1, every phase inside SpSn has a unique commit value (if any). Denote the phase k commit-value by C_k. By Lemma 1, every collect of $I_j(k,1)$ in phase k must see C_k. Consequently, all values proposed in all higher phases must contain C_k. It follows that for $k' > k$, if there exist a commit value $C_{k'}$ at phase k', then $C_{k'} \supseteq C_k$, and Containment follows.

We now prove property 2, the Speculation component of SpSn. We consider two clients p_i and $p_{i'}$, and assume that p_i returns at phase k from SpSn with return value Q_i, and $p_{i'}$ returns $Q_{i'}$ at a higher phase $k' > k$. By Containment, $Q_i \subseteq Q_{i'}$. We want to prove that $Q_i \in Y_{i'}$. Indeed, at the second step of phase k, both p_i and $p_{i'}$ collect the first value whose write into $I_j(k,2)$ completed. By assumption, p_i collects only the $(commit, Q_i)$ value, hence, $p_{i'}$ must see this commit-value and insert it to $Y_{i'}$ as needed.

Complexity of SpSn RW Solution

Our implementation of SpSn guarantees a return value Y with a **linear** number of configurations. This stems from the fact that only commit configurations are inserted into Y, and by Lemma 2, these configurations are related by containment, hence at most linear in the size of the set of proposals.

In terms of the number of primitive operations, $C.SpSn()$ contains multiple rounds of write-collect. More specifically, within a single invocation of $C.SpSn()$, the number of phases may be n, where n is the number of proposals. The number of individual write/read operations is $O(m * n)$, where m is the number of participating processes. In the message-passing implementation below (Section 4), the factor m is absorbed into the size of messages.

4 SpSn Message-Passing Solution

There exists a straight-forward message-passing emulation of the $C.SpSn$ RW solution above (Algorithm 1): Use an ABD SWMR emulation [3] by the servers-set C per each abstract register $I_i(k, step)$. Naively implemented, every write to each SWMR register incurs one exchange between the single-writer, the client p_i, and servers in C, and every read incurs two exchanges. Each phase in the

RW solution performs two steps, each does one write and m reads. Hence, the naive message-passing emulation takes $2 \times (2m + 1)$ majority-exchanges with C.

If we "open" the ABD emulation, we can easily see that there is no reason to iterate through the m registers one at a time. Instead, we can utilize two exchanges to bulk-read all registers. The message size will be proportional to the actual number of participants, which is m at worst, but in any real execution it may be much smaller than m. We can further optimize and coalesce some exchanges from different SpSn steps. Specifically, as evident from the proof of Lemma 1, it suffices for the registers to maintain regular semantics [8], not necessarily atomic. Therefore, we can omit the second exchange, the 'write-back', from read operations. The SpSn message-passing protocol resulting from all these improvements is depicted in Algorithm 2. It has a total of four exchanges between a client p_i and servers in C. More importantly, this protocol is *adaptive*, i.e., it removes the requirement to a priori know m.

With respect to correctness, because this message passing protocol simply "opens" the high-level shared-object abstractions, its correctness follows directly from the correctness of Algorithm 1.

Algorithm 2 *C.SpSn* message-passing protocol for client p_i and server $q \in C$

```
1: server q ∈ C local variables:
2:   | I(process, phase, step) → value, relation-map, initially empty
3:
4: client p_i local variables:
5:   | proposal, collect, commit, w
6:   | Y_i, initially ∅
7:
8: procedure SpSn(I_i)
9:   | client p_i:
10:  |   |                                                    ▷ initialization
11:  |   | proposal ← I_i
12:  |   | for k = 1, 2, 3, ... do
13:  |   |   |                                                ▷ first phase
14:  |   |   | commit ← true, collect ← ∅
15:  |   |   | send (C, write, p_i, k, 1, proposal) to all servers in C
16:  |   |   | wait for acknowledgments from a majority of C
17:  |   |   | send (C, read, p_i, k, 1) to all servers in C
18:  |   |   | for each reply w do
19:  |   |   |   | if w ≠ ∅ and w ≠ proposal then
20:  |   |   |   |   | collect ← collect ∪ w, commit ← false
21:  |   |   |                                                ▷ second phase
22:  |   |   | send (C, write, p_i, k, 2, ⟨commit, collect⟩) to all servers in C
23:  |   |   | wait for acknowledgments from a majority of C
24:  |   |   | send (C, read, p_i, k, 2) to all servers in C
25:  |   |   | for each reply ⟨w.commit, w.set⟩ do
26:  |   |   |   | if w.commit ≠ true then
27:  |   |   |   |   | commit ← false
28:  |   |   |   | if w.commit == true then
29:  |   |   |   |   | Y_i ← Y_i ∪ w.set
30:  |   |   |   | proposal ← proposal ∪ w.set
31:  |   |   |
32:  |   |   | if commit == true then
33:  |   |   |   | return proposal, Y_i
34:  |
35:  |   server q, on receipt of (C, write, p_j, k, step, value):
36:  |   | insert a relation (p_i, k, step) → value into I
37:  |   | send back acknowledgment to p_j
38:  |
39:  |   server, on receipt of (C, read, p_j, k, step):
40:  |   | send back all non-empty (·, k, step) values of I
41:  |
```

5 Dynamic Reconfiguration using SpSn

In this section, we use SpSn to manage configuration changes, which are expressed as a set of changes.

The core of procedure Propose(I_i) is very simple: Client process p_i invokes it with input I_i. It starts at the latest subscribed configuration. Propose() invokes SpSn, adopts the new configuration change Q_i, and repeats in every speculated configuration Y_i. This continues until the proposed configuration is the same as the output from SpSn. Then the client subscribes to it.

The only issue that somewhat compounds the treatment is a possible expiration of configurations. There are two ways in which a client may learn that its configuration subscription has been expired. The first is if an attempted SpSn fails. Recall that in our probem model (Section 2), we model this case as an exception raised during execution, indicating as cause a subscription of a new configuration that affected the expiration (line 4, Algorithm 3).

The second way is when a client p_i encounters a proposal by a client p_j which started Propose() with a newer configuration subscription. We model this case by annotating each proposal I_i at the beginning of Propose (line 8, Algorithm 3) with the starting configuration subscription, and denote it $I_i.start$. If p_i ever collects a proposal I_j whose $I_j.start$ indicates a later configuration subscription than $I_i.start$, then p_i starts over at $I_j.start$.

Algorithm 3 reconfiguration protocol at client p_i

```
1:  local variables:
2:  |  speculated, done, proposal
3:
4:  on exception "current configuration subscription expired by new configuration C":
5:  |  subscribe to C and start Propose over
6:
7:  procedure PROPOSE(I_i)
8:  |  I_i.start ← current configuration subscription
9:  |  speculated ← {I_i.start}, done ← ∅, proposal ← I_i
10: |  for U ∈ speculated \ done), in increasing containment order do
11: |  |  (Q_i, Y_i) ← U.SpSn(proposal)
12: |  |  done ← done ∪ {U}
13: |  |  proposal ← Q_i
14: |  |  if max_{I_j ∈ proposal} I_j.start is later than current subscription then
15: |  |  |  subscribe to max_{I_j ∈ proposal} I_j.start and start Propose over
16: |  |  speculated ← speculated ∪ Y_i
17: |  subscribe to proposal and return it
```

Correctness of Dynamic Reconfiguration protocol

The key insight driving the Dynamic Reconfiguration solution to convergence is that every configuration C has a unique successor that is guaranteed to appear in the output of every C.SpSn. We name the configuration *seed*, and define it formally as follows. Define $seed(C)$ as the commit configuration $I_j(k, 2)$ returned from C.SpSn as Q_j, whose phase k is the lowest for all returned Q_j. Inductively, define $seed^1(C) := seed(C)$, and $seed^{(i+1)}(C) := seed(seed^i(C))$. Intuitively, all that matters are seed configurations, since clients cannot skip them. The rest are mere inefficiencies, namely, speculated configurations traversed unnecessarily by clients due to the lack of consensus. This is the price of asynchrony.

Theorem 1. *If every live client p_i proposes one change in Propose(I_i), and then forever invokes Propose with an empty change, then eventually there is a time at which all living clients subscribe to the same configuration.*

Proof. Let U be a seed-configuration, and let p_i invoke $U.SpSn$ inside Propose() and return (Q_i, Y_i). It follows immediately from property 2 of SpSn (see Introduction) that $seed(U) \in Y_i$. Therefore, p_i invokes SpSn in $seed(U)$, and inductively in every $seed^i(U)$. Once new proposals cease to arrive, then starting from any seed configuration a client is subscribed to, the client will traverse all the seed configurations to the end of the succession.

Complexity of Dynamic Reconfiguration solution

The number of speculated configurations in Y_i output from C.SpSn to all clients is linear in the number of proposals, since they are related by Containment. Things are not so simple when we consider the Y_j sets returned by SpSn's in different configurations. Invoking SpSn in two different configurations, say $C.SpSn$ and $C'.SpSn$, might return $C.Y_j$ and $C'.Y_j$ containing items which are not related by containment. However, by negation, for every pair of speculations which are not related by containment, one must contain an input injected in a later subscription than the other. Hence, when a client encounters the later speculation (say D), it causes the client to subscribe to D and restart Propose in it. Therefore, the configurations actually traversed by clients (not the total ones ever held inside *speculated*) are ordered by containment. It follows that clients traverse in total a linear number of configurations in the number of different proposed changed.

6 Application: Read-Write Store

In this section we outline the design of a dynamic service, an elastic Read-Write store, built using Dynamic Reconfiguration. This service emulates a single, atomic multiple-writer multi-reader (MWMR) Read-Write register in our dynamic execution model. That is, in a dynamic store, our set *client* of client processes access a shared store service through the set *servers* of servers. The availability of servers for responding to Write and Read requests is governed by the client subscriptions to configurations. As in the Dynamic Reconfiguration problem, Propose() requests may occur independently and concurrently with Read and Write requests. The solution consists of three components:

- Inside Propose(), following every $U.SpSn$ a client needs to read the value stored at configuration U. At the end of Propose, the client writes the latest value with its original timestamp to the final configuration before it subscribes to it.
- To write a new value, a client first writes it into its current subscription configuration, and then invokes an empty Propose() in order to transfer the value into any new configuration subscription.
- To read the latest value, a client first invokes an empty Propose() and then returns the value it finishes with.

We now give the key insight for correctness. The key idea is that writing new information into subscription-configurations is done write-then-SpSn, while information gets transferred from one seed-configuration to the next by doing SpSn-then-read in each configuration. Consider a client p_i traversing through configuration C. If there is any write done in C, either the writer finished before the read, hence the read will see it. Or the writer's SpSn starts after the the reader's SpSn, hence see any reconfiguration proposal by the client. Finally, for any client not traversing through configuration C, there must exist some client which transferred information from C to a later configuration.

7 Conclusions

The germination of this paper is instructive. Being deeply invested in the idea that "behind any non-trivial distributed question there is a simple task," we asked ourselves what is the simple task behind the question in [2, 11]. We identified the task Speculating Snapshots (SpSn), and showed how previous solutions to the problem of reconfiguration solved the task, how we can solve the task in various models, and how to build a dynamic reconfiguration around it. Our parsimonious solution to the SpSn task in read-write wait-free drives a reconfiguration scheme linear in the number of intermediate configurations used, which is optimal. The number of operations may be subject to further optimization, in particular, using a BFS-like intermingling of SpSn's; this is left open for future work.

The problem tackled in this paper is fundamental to the dynamic nature of distributed systems. In distributed, mission critical settings, it is reasonable to assume that these dynamic changes occur slowly and allow to carefully migrate information in a changing system. This is the model assumed here. We already showed utility with a straw-man dynamic store design, and we envision that other dynamic services can be built equally easily.

More generally, in our solution to Speculating Snapshots (SpSn), we introduced a slight modification of Commit-Adopt. We expect that this new technique may become useful in other contexts.

Our work leaves open the question of operation complexity. Likewise, quantifying the relationship between real world scenarios and our slowly-changing fault model may be an interesting, practical challenge. Finally, we hope to employ this approach (identifying what is the **task** behind a problem) in other problems, as this experience shows promise.

Acknowledgments

We are thankful to Idit Keidar, Leslie Lamport and Alex Speigelman for helpful discussions. Part of this work was done when the first author visited MIT supported by National Science Foundation: CCF-1217921,CCF-1301926, and U.S. Department of Energy: DE-SC0008923.

References

1. M. Aguilera, I. Keidar, J.-P. Martin, and A. Shraer. Reconfiguring replicated atomic storage: A tutorial. *Bulletin of the EATCS*, (102):84–108, 2010.
2. M. K. Aguilera, I. Keidar, D. Malkhi, and A. Shraer. Dynamic atomic storage without consensus. *J. ACM*, 58(2):7:1–7:32, Apr. 2011.
3. H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, Jan. 1995.
4. G. Chockler, S. Gilbert, V. Gramoli, P. M. Musial, and A. A. Shvartsman. Reconfigurable distributed storage for dynamic networks. *J. Parallel Distrib. Comput.*, 69(1):100–116, Jan. 2009.
5. M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, Apr. 1985.
6. E. 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*, PODC '98, pages 143–152, New York, NY, USA, 1998. ACM.
7. S. Gilbert, N. Lynch, and A. Shvartsman. RAMBO II: Rapidly reconfigurable atomic memory for dynamic networks. In *Proceedings of International Conference on Dependable Systems and Networks*, pages 259–268, 2003.
8. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, Sept. 1979.
9. N. A. Lynch and A. A. Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *Proceedings of the 16th International Conference on Distributed Computing*, DISC '02, pages 173–190, London, UK, UK, 2002. Springer-Verlag.
10. P. Musial, N. Nicolaou, and A. A. Shvartsman. Implementing distributed shared memory for dynamic networks. *Commun. ACM*, 57(6):88–98, June 2014.
11. A. Shraer, J.-P. Martin, D. Malkhi, and I. 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.
12. J. L. Welch and H. Attiya. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. McGraw-Hill, Inc., Hightstown, NJ, USA, 1998.