SmartMerge: A New Approach to Reconfiguration for Atomic Storage

Leader Jehl, Roman Vitenberg, Hein Meling

To cite this version:

Leader Jehl, Roman Vitenberg, Hein Meling. SmartMerge: A New Approach to Reconfiguration for Atomic Storage . DISC 2015, Toshimitsu Masuzawa; Koichi Wada, Oct 2015, Tokyo, Japan. 10.1007/978-3-662-48653-5_11 . hal-01206174

HAL Id: hal-01206174

https://hal.science/hal-01206174

Submitted on 29 Sep 2015

SmartMerge: A New Approach to Reconfiguration for Atomic Storage

Leander Jehl[1], Roman Vitenberg[2], and Hein Meling[1]

[1] Department of Electrical Engineering and Computer Science,
University of Stavanger, Norway
[2] Department of Informatics, University of Oslo, Norway

Abstract. In this paper, we study reconfiguration mechanisms for atomic storage systems. We observe that the state of the art approach for reconfiguration in an asynchronous environment has several disadvantages compared to the classical consensus-based approach, which requires eventual synchrony. For example, an unfortunate combination of remove operations may lead to a configuration with too few or even no processes. We present SmartMerge, a novel approach that provides most of the benefits of consensus-based reconfiguration, yet can be implemented in a fully asynchronous system. SmartMerge utilizes a merge function to aptly combine concurrently issued changes to both the set of processes and the quorum system of the storage. The approach is general and can use any suitable function.

In addition to the expressive reconfiguration policies enabled by SmartMerge, our atomic storage also has improved efficiency: Every reconfiguration imposes only a constant overhead on concurrent read and write operations.

Keywords: Atomic Storage, Reconfiguration, Asynchronous System

1 Introduction

In the age of cloud computing, an abundance of compute resources with different capabilities are available at data centers across the globe. These data centers deploy a variety of services replicated for fault-tolerance. It is typical for the administrators of the data center to update both the composition of machines in the data center and the composition of replicas running a service, because of the need to regularly upgrade the machines, replace failed components, and accommodate for changes in the service load. Such reconfiguration operations are rather frequent in practice as evident, e.g., from the traces of a Google data center [1].

One of the main challenges of supporting reconfiguration is to ensure consistency when multiple users submit concurrent requests. A monitoring component can be tracking software and hardware failures, upgrades, and load of queries and updates to the replicas. Acting upon this information, it may be issuing requests autonomously, without human intervention [2]. It is envisioned that many such components may be deployed in a large-scale data center at the same time, which may result in multiple concurrent uncoordinated and even conflicting requests for reconfiguration.

The traditional approach to resolve this situation is to use consensus to choose one of the proposed configurations, see e.g. [3]. The proposal for a new configuration in this scheme would include a desired set of replicas along with a quorum system to use. The main disadvantage of this consensus-based approach is that its liveness is impossible to guarantee in asynchronous systems characteristic of large-scale data centers.

In order to address this issue, an alternative conceptual approach proposed in the DynaStore system [4] is to provide the users with an interface to request incremental additions or removal of processes. In case of commutative concurrent requests, the system directly combines the changes instead of choosing just one of them. Furthermore, since changes commute, they do not need to be ordered. This allows the approach to be implemented in a fully asynchronous system, without relying on eventual synchrony or leader election to solve consensus. Henceforward we refer to this conceptual approach as *DirectCombine*.

Another advantage of DirectCombine over the consensus-based approach is that non-conflicting changes can be proposed concurrently. For example, two changes, one removing replica a and one replacing replica b with e, can be issued concurrently by different processes and are both realized in the resulting configuration. Using the consensus-based approach, only one of these requests would be chosen and applied.

However, the fact that all proposed changes are applied can also lead to problems. If two proposed reconfigurations are trying to remove a different replica each, applying both removals may incidentally result in a configuration with a small number of replicas and thus, low fault-tolerance. In the extreme case, a combination of several removals may result in an empty configuration, in which no further operations can be performed.

It might be preferable for a system to abort the reconfiguration process than to switch to such a configuration. In general, configurations with too few or too many processes, or with an unfavorable distribution of processes across data centers can be *unacceptable* in practice, because of low fault tolerance, high network latencies, or administrative restrictions. Using the consensus-based approach, the system can only be reconfigured to an unacceptable configuration, if such a configuration was proposed. Thus it is easy to avoid these configurations.

Another disadvantage of the automatic combination of different reconfiguration requests in DirectCombine is the need to autonomously recompute the quorum system on the fly. While such dynamic computation is simple for majority quorums, it may not be feasible in the general case or in real-world situations. In heterogeneous systems spanning over multiple data centers and a complex network topology, quorum systems may have topology-induced structure. Furthermore, adjusting weights and the balance between read and write quorums can be the key to meeting service level agreements under dynamic load patterns. It is therefore undesirable to limit these systems to majority quorums.

Table 1. Comparison of reconfiguration approaches

	Avoids unacceptable configurations?	Can combine multiple proposals?	Can easily switch quorum system?	Asynchronous system
Consensus-based	yes	no	yes	no
DirectCombine	no	yes	no	yes
SmartMerge	yes	yes	yes	yes

In this paper we present a novel approach called SmartMerge. In the core of the approach lies a SmartMerge function that intelligently combines different, concurrently issued reconfiguration requests. The approach based on such a function has several advantages:

Generalized interface for the reconfiguration operation, that instead of just adding and removing one specific process can operate with rules and policies to change the number of replicas, add several replicas as a group, set weights or priorities to individual potential replicas, or introduce any correlation between replicas. Such policies allow the SmartMerge function to produce meaningful resulting configurations in the case of concurrent divergent requests, or to configure the set of replicas automatically in presence of failures.

Easy switching between different quorum systems: For example, a replication scheme can employ write-all read-one quorums, to minimize the latency of parallel reads, while switching to majority quorums before upgrades are performed, or whenever failures and temporary outages are expected.

Definition and avoidance of unacceptable configurations as part of the policy: In the simplest case, it is possible to define the minimum number of replicas and maintain it across all reconfigurations and failures.

Similarly to DirectCombine, SmartMerge can be implemented in an asynchronous system. We summarize the pros and cons of the three approaches in Table 1.

The SmartMerge function can be tailored to the specifics of the service, replication scheme, data center and its topology, and many additional factors. We show a concrete example of such a function in Section 2, in order to illustrate the aforementioned features of the approach.

We apply our approach to reconfiguration of atomic storage. Atomic storage is a key problem in distributed systems that can be implemented in an asynchronous system [5]. Both the consensus-based approach [3] and DirectCombine [4] have been applied to this problem.

The main contribution of this work is that we show how SmartMerge can be implemented for atomic storage in an asynchronous system. This is done in a generic way parametrized by an externally defined SmartMerge function, such as the one presented in Section 2.

The key idea behind our implementation is as follows: Under the assumption that the SmartMerge function is commutative, associative, and idempotent, it induces a lattice of all possible reconfiguration requests. If finitely manyconcurrent requests are proposed, we ensure that eventually every process will adopt the merge (i.e., a lattice join) of all these competing requests as its configuration.

To implement wait-free reads and writes, these operations cannot wait until a reconfiguration has completed. A common solution to this problem is to read from or write to the old, the new and all possible intermediate configurations, while reconfigurations are ongoing. However, before reaching the lattice element that is the join of all concurrent requests, the service can intermediately adopt a join of any subset of these requests. The number of such joins in the lattice is exponential with the number of competing requests. To avoid contacting processes in all these configurations during read and write operations, we submit proposed requests to lattice agreement, which returns elements from a totally ordered subset of the lattice, including the greatest element. We use only elements, that were returned by lattice agreement as intermediate configurations. The number of different configurations returned by lattice agreement during an execution is at most the number of changes proposed during this execution, making it feasible to read from or write to all intermediate configurations. Additionally, if one of these configurations is adopted by the service, operations need no longer read from or write to elements smaller than this configuration.

To read a value from the register while changes are applied to the configuration, we read the register values stored in all configurations returned by lattice agreement and return the most recent one. Similarly, to write a value, we write to all these configurations. Read and write operations do not participate in lattice agreement, but only contact processes in configurations returned by lattice agreement. We show that in an execution where at most r changes are proposed, all read and write operations cumulatively contact processes in at most $r + 1$ different configurations. A single read or write operation contacts the processes in any configuration at most twice. This gives a bound on the latency of read and write operations that is the same as for the consensus-based approach in [3], but a significant improvement compared to DynaStore [4].

2 System Model

We assume a possibly infinite set of processes Π, communicating via asynchronous channels. Each process $\mathbf{p} \in \Pi$ has a unique identifier $\mathbf{p}.id$.

Processes can fail at any time during an execution by stopping to take any actions. We assume that messages are not corrupted and that, if two processes do not fail during an execution, all messages sent between these processes are eventually delivered.

Not all processes in Π need to be known a priori. We therefore maintain a finite set of available processes $\mathcal{A} \subset \Pi$. Newly discovered processes are added to $\mathcal{A}$ by a reconfiguration operation. We also maintain a set $\mathcal{A}_{rm}$ which contains processes that are no longer available and have been removed from $\mathcal{A}$. If two processes disagree whether $\mathbf{p}$ is available, they can use $\mathcal{A}_{rm}$ to determine if one process missed to add $\mathbf{p}$ or the other missed out on removing $\mathbf{p}$. Thus two processes using $\mathcal{A}^1, \mathcal{A}^1_{rm}$ and $\mathcal{A}^2, \mathcal{A}^2_{rm}$ respectively, can combine their knowledge and both switch to using $\mathcal{A} = (\mathcal{A}^1 \backslash \mathcal{A}^2_{rm}) \cup (\mathcal{A}^2 \backslash \mathcal{A}^1_{rm})$ and $\mathcal{A}_{rm} = \mathcal{A}^1_{rm} \cup \mathcal{A}^2_{rm}$ instead. While it is impossible to distinguish a faulty process from a slow process in an asynchronous system, recent works have proposed failure detectors that reliably detect all [6] or at least some failures [7], without relying on timing assumptions. The possibility to remove processes from $\mathcal{A}$ allows our service to be used together with a reliable, unreliable or no failure detector.

Only a subset $\mathcal{P} \subset \mathcal{A}$ of the available processes is actually running the service. These are organized in a *service configuration.* A service configuration c is a tuple $(\mathcal{P}_c, \mathcal{WQ}_c, \mathcal{RQ}_c)$, where $\mathcal{P}_c \subset \Pi$ is a finite set of processes, and $\mathcal{WQ}_c$ and $\mathcal{RQ}_c$ are collections of subsets of $\mathcal{P}_c$, called read and write quorums, such that any read quorum from $\mathcal{RQ}_c$ intersects with every write quorum from $\mathcal{WQ}_c$. We write $\mathfrak{C}$ for the domain of all such configurations.

Given the set $\mathcal{A}$ of available processes, the choice of a service configuration is determined by a policy. We model such a policy as a tuple $(\mathit{srvConf}(), \mathit{info})$, where $\mathit{srvConf}$ is a function $\mathcal{P}_f(\Pi) \rightarrow \mathfrak{C}$ that maps any finite subset of Π to a service configuration. info is auxiliary information describing how the policy is combined with other policies. We write $\mathcal{PL}$ for the set of all policies $(\mathit{srvConf}(), \mathit{info})$ that can appear in an execution.

A reconfiguration can both add and remove available processes and propose a new policy. We say that a reconfiguration proposes a *blueprint* for a service configuration. We express a blueprint as a tuple $(\mathcal{A}, \mathcal{A}_{rm}, \mathit{policy})$, with finite sets $\mathcal{A}, \mathcal{A}_{rm} \subset \Pi$ and $\mathit{policy} \in \mathcal{PL}$. We write $\mathfrak{R}$ for all such tuples. Applying $\mathit{policy}.\mathit{srvConf}(\mathcal{A})$ results in a service configuration that includes only available processes and satisfies the rules expressed by policy. We say that a blueprint $r = (\mathcal{A}, \mathcal{A}_{rm}, \mathit{policy})$ determines the service configuration $c = \mathit{policy}.\mathit{srvConf}(\mathcal{A})$.

Lattice and order of blueprints. For SmartMerge, we require that the system manager provides a commutative, associative and idempotent function, join, to merge policies. These properties are quite intuitive in practice, as it can be seen in the example later in this section. We merge blueprints by combining $\mathcal{A}$ and $\mathcal{A}_{rm}$ as described above and use join to combine the policies. Collectively, this defines a function $\mathit{merge}(\mathfrak{R}, \mathfrak{R}) \rightarrow \mathfrak{R}$ that combines blueprints. We assume an initial element $r_I \in \mathfrak{R}$, such that $\forall r \in \mathfrak{R} : \mathit{merge}(r, r_I) = r$ holds.

Since join is commutative, associative and idempotent, these properties also hold for merge. A set $\mathfrak{R}$, together with the merge function, thus is an algebraic semi-lattice [8]. Again, due to these properties we can write $\mathit{merge}(\{r_1, r_2, ...\})$ instead of $\mathit{merge}(r_1, \mathit{merge}(r_2, ...))$.

This lattice $(\mathfrak{R}, \mathit{merge})$ is bounded by the initial element r_I. In the remainder of this work, we simply write *lattice* instead of bounded semi-lattice. Due to its properties, merge defines a partial ordering $\sqsubseteq$ on $\mathfrak{R}$ by the relation $\forall r_1, r_2 \in \mathfrak{R} : r_1 \sqsubseteq r_2 \Leftrightarrow \mathit{merge}(r_1, r_2) = r_2$. We write $r_1 \sqsubset r_2$ for $(r_1 \sqsubseteq r_2 \wedge r_1 \neq r_2)$ and say that r_2 is a *greater blueprint* (as lattice element) than r_1 and that r_1 is a *smaller blueprint.* We write $r_1 \not\sqsubseteq r_2$ for the negation of $r_1 \sqsubseteq r_2$. Note that $r_1 \not\sqsubseteq r_2$ is not equivalent to $r_2 \sqsubset r_1$, since $\sqsubseteq$ is only a partial order.

The properties of merge imply that for any $r_1, r_2 \in \mathfrak{R}$, $r_1 \sqsubseteq \mathit{merge}(r_1, r_2)$ holds. A process can test whether $r_1 \sqsubseteq r_2$ holds by comparing r_2 and $\mathit{merge}(r_1, r_2)$.

In our algorithm in Section 4, we use this to compute the minimal or maximal element in a set of comparable blueprints.

Lattice Agreement Service GLA. As mentioned in the introduction, if reconfigurations propose different blueprints we want to reconfigure to the merge of all proposed blueprints. We use an external generalized lattice agreement service (GLA) for this. GLA offers an operation **la-propose**(r), that takes a blueprint $r \in \mathfrak{R}$ as argument and returns another blueprint r', such that the following properties hold. These properties imply that the merge of all input values is among the returned values.

Validity A returned value is the merge of inputs to **la-propose** operations.
Monotonicity An operation **la-propose**(r) returns r' such that $r \sqsubseteq r'$.
Comparability Any two values returned by **la-propose** are comparable with respect to $\sqsubseteq$.

GLA can be easily implemented using the algorithm specified in [9]. In generalized lattice agreement on a lattice $(\mathfrak{R}, \mathit{merge}, \sqsubseteq)$, processes can receive values $v_i \in \mathfrak{R}$ from clients. The processes then learn a sequence of values $w_0 \sqsubseteq w_1 \sqsubseteq ...$ such that validity and comparability hold for learned values. Further, if a value v_i is received at a correct process, every correct process eventually learns a value w_j, such that $v_i \sqsubseteq w_j$ holds. The **la-propose**(r) operation can be easily implemented by sending r to all processes running generalized lattice agreement and returning some value r' learned by any of these processes, for which $r \sqsubseteq r'$ holds. The complexity of the algorithm, as presented in [9] adapts to the number of values actually proposed. Thus, if **la-propose** is invoked only r times during an execution, every invocation will return after at most $\mathcal{O}(r)$ steps.

Some works using the consensus-based approach assume an external configuration manager that receives reconfiguration requests and chooses a sequence of configurations (e.g. [3]). Different from these works our GLA can be implemented in an asynchronous system and does not need consensus. However GLA only returns comparable elements without sequence numbers. Thus, if r and r' have been returned by two **la-propose** invocations to process p, and $r \sqsubset r'$ holds, it is impossible for p to determine if some other value $\hat{r}$ for which $r \sqsubset \hat{r} \sqsubset r'$ holds, has been returned by another **la-propose** invocation to a different process. We show in this paper that the weaker guarantees of GLA are still sufficient to implement a reconfigurable atomic register.

Example. We now give a more detailed example of a merge function, that illustrates the use of policies, easy switching between different quorum systems, and avoidance of unacceptable configurations, which are defined as configurations with fewer than k processes. In this example, policies are determined by a set of rules, shown in Table 2. One reconfiguration can change several of these rules.

We can use $\mathit{addMan}(p)$ to mark a specific process p as mandatory element of $\mathcal{P}$. Similarly, we can specify a process p as optional using $\mathit{remMan}(p)$. Once marked as optional, a process can no longer be marked as mandatory. The reason for this is explained in Section 3. Additionally, we can specify a desired size for

Table 2. Rules for building a configuration, supported by our example

Rule	Effect	Rule	Effect
addMan(p)	mark p as mandatory	*setSize*(n)	specify desired size n for $\mathcal{P}$
remMan(p)	mark p as optional	*majority*()	use majority quorums
		waro()	use write-all-read-one quorums

$\mathcal{P}$ using *setSize*(n). If the number of mandatory processes is fewer than the desired size, the policy function adds additional processes from $\mathcal{A}$. Finally we can use *majority*() and *waro*() to specify whether the quorum system should use majority or write-all-read-one (WARO) quorums. For WARO quorums we simply set $\mathcal{WQ}_c = \{\mathcal{P}_c\}$ and $\mathcal{RQ}_c = \{\{\mathbf{p}\}|\mathbf{p} \in \mathcal{P}_c\}$. For majority quorums any subset containing at least a majority of the processes in $\mathcal{P}_c$ forms a write quorum ($\mathcal{WQ}$), while any subset containing at least half the processes is a read quorum ($\mathcal{RQ}$). When the size or quorum system is changed, using *setSize*(n), or *majority*() and *waro*(), the policy info has to specify an epoch number that is used in the combination function.

When combining policies, we differentiate between processes explicitly marked as optional, and unmarked processes. Combining two policies, a process explicitly marked as optional in one of the policies retains this marking. Processes marked as mandatory in one of the policies and not explicitly marked as optional remain mandatory. When two policies include different *setSize* rules or specify different quorum systems we adopt the size and quorum system from the policy with the higher epoch number. If epoch numbers are equal, we choose the larger *size* and majority quorums, if present, since these choices provide higher fault tolerance.

Using the rules from this example, $|\mathcal{P}| \geq k$ holds as long as *setSize*(n) with $n < k$ is invoked and there are at least k processes available.

3 Problem: Atomic Storage using Smart Merge

In this section we specify our reconfigurable multi-reader multi-writer atomic register. We assume a set of possible register values $\mathcal{V}$, and a lattice of blueprints $(\mathfrak{R}, \sqsubseteq, \mathit{merge})$ with minimal element r_I. We provide three operations, **read**, **write** and **reconf**. A **read**() operation returns either a value $v \in \mathcal{V}$ or $\perp \notin \mathcal{V}$. A **write**(v) operation takes an input $v \in \mathcal{V}$.

We require that **read** and **write** operations are linearizable [10], and that in a sequential execution, every **read** returns the value of the last **write**, or $\perp$ if no **write** occurred before the **read**. This is the standard safety property of atomic registers. The liveness of **read** and **write** depends on the reconfigurations invoked. A reconfiguration changes which processes may fail, but also which processes should invoke operations. We therefore discuss the **reconf** operation, before presenting a common liveness property for all operations.

A **reconf**(r) operation proposes a blueprint $r \in \mathfrak{R}$, and returns a value $r' \in \mathfrak{R}$ that determines the service configuration of the register. We say that a blueprint r is *chosen* before time t in an execution, if r was returned by a **reconf** operation before t in that execution. We say that r_I is chosen by an implicit **reconf** operation, at the beginning of any execution. The following safety properties govern which values may be chosen. These properties and other concepts introduced in this section are defined in the context of a single execution:

Validity A chosen value r' is the merge of input values to **reconf** operations.
Monotonicity If r' is chosen by **reconf**(r), then $r \sqsubseteq r'$ holds.
Comparability Any two chosen values are comparable, with respect to $\sqsubseteq$.
Stability If r was chosen before the invocation of **reconf**, which returns r', then $r \sqsubseteq r'$ holds.

Validity, monotonicity and comparability are standard requirements for values returned from a lattice (e.g. in lattice agreement [9], [11]). However, to our knowledge we are the first to require these properties in the context of reconfiguration. Validity ensures that no arbitrary value is chosen. Monotonicity implies that r' is the merge of r with another blueprint, e.g. $r' = \mathit{merge}(r, r')$. This implies that rules introduced in r are also applied in r'. For example, a process mandatory in r will also be mandatory in r', unless it was explicitly removed by another **reconf** operation.

Our goal is for our service to eventually use a single service configuration. This can be accomplished since comparability implies that at any time t, there exists a blueprint among those chosen before t, that is maximal with respect to $\sqsubseteq$. We call this the *current blueprint* at time t, and the service configuration, determined by this blueprint is called the *current configuration.*

To change the current blueprint, a **reconf** operation has to choose a blueprint that is a greater lattice element than all previously chosen blueprints. Thus the service can only replace r_1 with r_2 if $r_1 \sqsubseteq r_2$ holds. It is therefore not possible to add a process to $\mathcal{A}$ after it has been removed, since it will be listed in $\mathcal{A}_{rm}$. Similarly for our example above, adding $\mathit{addMan}(p)$ after $\mathit{remMan}(p)$ will not result in any changes. In practice however, a process can be re-added with a different identifier.

Stability allows us to use the **reconf** operation to read the current blueprint. In our example in Section 2, this can be used to determine the current epoch number. Stability also implies that a new blueprint will always be merged with a previously chosen blueprint. Thus, the input to a **reconf** operation does not need to specify all desired rules and available processes. It is enough to include all new rules and processes relative to some previously chosen blueprint.

We now specify which processes need to be correct to guarantee liveness. We say that a process is *correct* at time t, if it did not fail before t. A service configuration is *available* at time t, if there exists a read and write quorum of this service configuration, such that all processes in these quorums are correct at t. We require that the current configuration is available. To allow state transfer during a reconfiguration, we also require that the service configurations of any new blueprints are available. This is a common requirement for reconfigurations [3]. For SmartMerge, we define that r is a *candidate blueprint* at time t, if it is a possible return value for some outstanding **reconf** operation at time t, and it is greater than the current blueprint ($cur \sqsubseteq r$). A service configuration determined by a candidate blueprint, is called a *candidate configuration*.

We only require liveness for operations invoked by a process currently running the register. These processes are called active: A process $\mathbf{p}$ is *active* if it is correct at all times and after some time t, $\mathbf{p} \in \mathcal{P}_{cur}$ always hold for the current configuration *cur*. This definition is similar to [4]. We could also include a larger set of clients, which can invoke operations, similar to [3] or [12]. This adds no significant challenges to the problem and we omit it due to space constraints.

The following property summarizes under which conditions an operation is required to return:

Liveness Suppose that only finitely many **reconf** operations are invoked during an execution, and at any time the current and all candidate configurations are available. Then a **read**, **write** or **reconf** operation, invoked by an active process will return.

It was established in [13] that even a regular register is impossible to implement, if the configuration changes infinitely often. Thus, we assume that only finitely many reconfigurations are invoked.

Once $cur \not\sqsubseteq r$ holds for a blueprint r and the current blueprint *cur*, then r can no longer become the new current blueprint. Thus r can be discarded. We say that r is *outdated*. According to the definition of the current blueprint, r is outdated, when some **reconf** operation returned r', such that $r' \not\sqsubseteq r$ holds. A process can easily test the condition $r' \not\sqsubseteq r$ by computing $merge(r', r) \neq r$.

According to liveness, a process $\mathbf{p}$ can stop once it can no longer become part of the current configuration. In our example, this is the case once $\mathbf{p} \in \mathcal{A}_{rm}$ holds for the current blueprint.

To our knowledge we are the first to propose a scheme that determines which blueprints are outdated and which processes can stop, based on return values of reconfigurations. RAMBO [3] uses a garbage collection mechanism to find outdated configurations. Thus it potentially takes longer to detect outdated configurations. Furthermore, it is not possible to determine which configurations are outdated, based on return values of reconfiguration operations. The specification of DynaStore [4] does not use configurations, thus no outdated configurations or blueprints are defined. Instead, a process can stop as soon as its removal is proposed. To guarantee liveness DynaStore has to restrict the number of concurrent removals. This poses a significant restriction on reconfigurations, e.g. it disallows concurrent replacement of all processes with new ones.

4 Algorithm: Atomic Storage using Smart Merge

We now present our implementation of a reconfigurable atomic register. The implementation consists of a support for **reconf** operations (Algorithm 2) and two functions **get** and **set** (Algorithm 3) that access the state of the register. Algorithm 2 and Algorithm 3 internally use a configuration object to contact the processes and access the state stored in a service configuration. We present the implementation of a configuration object in Algorithm 1. The **set** and **get** functions mask concurrent reconfigurations, so that we can use them to implement regular or atomic registers, using standard algorithms designed for a single configuration (e.g. [5], [14]). For completeness, we show an implementation of such atomic reads and writes in Algorithm 4.

The highlights and new techniques of our implementation include the use of the lattice agreement abstraction in Algorithm 2 and handling of the returned values. It makes the algorithm significantly more efficient, since the use of generalized lattice agreement reduces the number of configurations that have to be processed. Furthermore, we implement **set** and **get** without relying on a sequence of chosen configurations, using only the weaker guarantees provided by GLA. Finally, in our implementation, the removal of outdated blueprints and their configuration objects is gracefully and efficiently integrated with concurrent operations.

Our reconfigurable register relies on a configuration object C (Algorithm 1), which includes the service configuration c, the register state S, and a set $\mathit{next} \subset \mathfrak{R}$. A *register state* $S \in \mathcal{V} \times \mathcal{T}$ is a pair, consisting of a register value $v \in \mathcal{V} \cup \{\bot\}$ and a timestamp $ts = (n, id) \in \mathcal{T}$. A timestamp consists of a sequence number $n \in \mathbb{N}$ and a process identifier id. Timestamps are ordered lexicographically. *next* holds a set of blueprints whose purpose we explain below.

The configuration object C also abstracts communication between the processes in $\mathcal{P}_c$ through a set of regular registers. For every process $\mathbf{p} \in \mathcal{P}_c$, C contains registers $\mathbf{p}.S$ and $\mathbf{p}.\mathit{next}$ with the state of $\mathbf{p}$'s local variables. Only $\mathbf{p}_i$ can write to $\mathbf{p}_i.S$ and $\mathbf{p}_i.\mathit{next}$, but they can be read by all processes in $\mathcal{P}_c$. To read the register state of C, a process invokes $C.\mathit{readS}()$, which reads all registers $\mathbf{p}.S : \mathbf{p} \in \mathcal{P}_c$ and returns the state with the highest timestamp. To read the set of next blueprints, a process reads all registers $\mathbf{p}.\mathit{next}$ and returns the union of these values. Finally, we use reliable broadcast (rb), to notify members of a new configuration when it becomes the current configuration. If some process completes $C.\mathit{rb}.\mathit{broadcast}(m)$, and a quorum of processes in $\mathcal{P}_c$ do not fail, all correct processes in $\mathcal{P}_c$ will eventually invoke $C.\mathit{rb}.\mathit{deliver}(m)$.

Regular registers and reliable broadcast can be implemented using textbook algorithms [15], designed for an asynchronous message passing system with a known finite set of processes and a fixed quorum system. Our configuration object encapsulates the processes and quorums of a static configuration, which forms the system on which these algorithms operate. To use a different configuration, we create a new object. If a configuration becomes outdated, its processes might stop and the static algorithms that operate on this configuration might never return. We therefore abort any method in an outdated configuration.

In our algorithm, we use the **la-propose** primitive specified in Section 2. We say that a value returned by **la-propose** is *learned*. Every process maintains a set L, that contains the current and some candidate blueprints (Algorithm 2). All elements in L were learned from **la-propose**. Comparability for **la-propose** implies that L is totally ordered by $\sqsubseteq$. For every blueprint $r \in L$, we also store a configuration object $C[r]$ determined by r. $C[r]$ is created or deleted, when

Algorithm 1 Configuration object C at process $\mathbf{p}_i$

```
1:  State :
2:      $c : (\mathcal{P}_c, \mathcal{RQ}_c, \mathcal{WQ}_c)$                      {Service configuration}
3:      $S : (S.v, S.ts) \leftarrow s_0$          {Register state $S \in \mathcal{V} \times \mathcal{T}$, $s_0 = (\perp, \mathbf{p}_i.id)$}
4:      $next \leftarrow \emptyset$                                  {Next blueprints, $next \subset \mathfrak{R}$}
5:  Communication Abstractions:
6:      Regular SWMR registers
7:          for each $\mathbf{p} \in \mathcal{P}_c : \mathbf{p}.S : (S.v, S.ts)$     {storing $S \in \mathcal{V} \times \mathcal{T}$, initially $s_0$}
8:          for each $\mathbf{p} \in \mathcal{P}_c : \mathbf{p}.next$             {storing $next \subset \mathfrak{R}$, initially $\emptyset$}
9:      $rb$                                                     {Reliable broadcast}
10: $readS()$
11:     for $\mathbf{p} \in \mathcal{P}_c$
12:         $s_\mathbf{p} \leftarrow \mathbf{p}.S.read()$
13:     $t \leftarrow \max\{s_\mathbf{p}.ts | \mathbf{p} \in \mathcal{P}_c\}$
14:     return $s_\mathbf{p}$: s.t. $\mathbf{p} \in \mathcal{P}_c \wedge s_\mathbf{p}.ts = t$
15: $writeNext(target)$                  {invoked by $\mathbf{p}_i$}
16:     $next \leftarrow next \cup \{target\}$
17:     $\mathbf{p}_i.next.write(next)$
18: $readNext()$
19:     for $\mathbf{p} \in \mathcal{P}_c$ do
20:         $C_\mathbf{p} \leftarrow \mathbf{p}.next.read()$
21:     return $\bigcup\{C_\mathbf{p} | \mathbf{p} \in \mathcal{P}_c\}$
22: $writeS(s)$                          {invoked by $\mathbf{p}_i$}
23:     if $s.ts > S.ts$ then
24:         $\mathbf{p}_i.S.write(s)$
25:         $S \leftarrow s$
```

r is added or removed from L. $C[r]$ can also be created, when it is accessed remotely though its communication abstractions. However, accesses to objects, that belong to outdated blueprints are ignored.

We now discuss the **reconf** operation shown in Algorithm 2. The operation starts by passing the blueprint rr to **la-propose**. We add the value learned from **la-propose** to L. Since L is totally ordered by $\sqsubseteq$, we can choose the maximum in L, as *target* blueprint for our reconfiguration. To ensure that other processes know that *target* was learned, we write *target* to all configurations that were created using elements of L (Line 11). We also read the register state in all these configurations to collect an up-to-date state (Line 12). On Line 14 we invoke $C[fr].readNext()$, to find other learned blueprints and add them to L. Since we assign elements from L to fr in order (Line 16), and $C[fr].readNext()$ only returns blueprints larger than fr, these new blueprints will be processed later.

Validity and comparability already hold for *target* on Line 6. To ensure stability, we replace *target* with a larger learned value on Line 8, if possible. After processing all elements from L, we transfer the register state with the highest timestamp, that was read, to $C[target]$ (Line 18).

Before returning, and thus choosing *target*, we broadcast a $\langle$Chosen, *target*$\rangle$ message to all processes in $C[target].\mathcal{P}_c$ using the reliable broadcast (Line 19).

The processing of a $\langle$Chosen, *target*$\rangle$ message is shown in Algorithm 5. We ignore a Chosen message for a blueprint smaller than *cur*. If $\langle$Chosen, *target*$\rangle$ was sent, some process completed state transfer to *target*. We can therefore remove all elements smaller than *target* from L. Finally, if *target* was returned by a **reconf** operation, smaller blueprints $r \sqsubset target$ are outdated, according to our definition in Section 3. Thus $C[r]$ might no longer be available. In this case, it may become impossible to read from or write to the registers in $C[r]$. We therefore abort all current and future methods on $C[r]$. On abort, the write methods *writeNext*, *writeS* and *rb.broadcast* simply return, while *readNext* and *readS* return the default values ($\emptyset$ and s_0).

We next present our **set** and **get** functions used to read and write the register state (value, timestamp) from the current configuration. They are shown in Algorithm 3. **set** writes a register state to all configuration objects $C[r]$ that were created using $r \in L$, while **get** reads the register state in all these configurations, and returns the one with the highest timestamp. Note that when using $\mathit{writeS}(s)$, the register state is only overwritten if its timestamp is smaller than $s.ts$. **set** and **get** also invoke *readNext* to add new learned values to L.

Algorithm 4 shows a possible implementation of atomic reads and writes using **set** and **get**. Note that on Line 7, we create a unique timestamp by increasing the sequence number returned by **get**, and adding the writer's identifier $\mathbf{p}_i.id$.

Discussion. We say that a blueprint r is *used* in an operation if the operation invokes methods on $C[r]$. To analyze the overhead concurrent **reconf** operations impose on **read** and **write** operations, we first establish the maximum number of blueprints and configuration objects used in one operation. An operation only uses blueprints from L, which only holds the initial element r_I and values learned from GLA (Algorithm 2, Lines 6, 7). In an execution with r **reconf** operations, at most r different values are learned. Thus all operations use at most $r + 1$ different blueprints.

In DynaStore [4], the only other reconfigurable atomic register using a purely asynchronous system, a single operation may have to contact processes in many different configurations. Reconfigurations in DynaStore do not invoke lattice agreement. Instead a configuration uses a weak snapshot object to store the set of next configurations *next*. Without lattice agreement, non-comparable values can be written to *next*. Therefore a process in DynaStore adopts the merge of all configurations, returned by *readNext* as new target configuration. If three reconfigurations, with target configurations c_x, c_y and c_z are started concurrently, they can all be written to $r_I.next$. A concurrent **read** operation might not only have to contact c_x, $\mathit{merge}(c_x, c_y)$ and $\mathit{merge}(c_x, c_y, c_z)$, but also c_y and c_z. Additionally, **read** operations in DynaStore also write to *next*. Thus, a **read** operation concurrent with the three reconfigurations above might read $\{c_x, c_z\}$ from $r_I.next$, and thus write $c_{xz} = \mathit{merge}(c_x, c_z)$ to $c_x.next$. This creates another configuration that other **read** operations have to contact. If $2^{r-1} - r$ **read** operations and r reconfigurations are invoked concurrently, each by a different process, one of these reads might have to contact as many as $2^{r-1} + r$ configurations.

We now analyze the number of communication steps for **read** and **write** operations. Note that, the different registers that read in a *readS* or *readNext* method, can be read concurrently (see Algorithm 1). It is even possible to perform the methods $C[gt].\mathit{readS}$ and $C[gt].\mathit{readNext}$ on Lines 4 and 6 of Algorithm 3 concurrently, reading $\mathbf{p}.S$ and $\mathbf{p}.\mathit{next}$ at the same time. Thus a **get** function only requires one communication step per used blueprint. Similarly, we can perform the methods $C[st].\mathit{writeS}$ and $C[st].\mathit{readNext}$ invoked in the **set** function concurrently. Since both **set** and **get** use at most $r+1$ blueprints, a **read** or **write** operation requires at most $2r+2$ communication steps. This bound is the same as for the consensus-based approach in RAMBO [3].

Algorithm 2 Register Reconfiguration

```
State :
    $L \leftarrow \{r_I\}$                                  {Ordered set of learned, not outdated blueprints}
    $\forall r \in L : C[r] = CO(c : r.policy.srvConf(r.\mathcal{A}))$   {Conf. object, determined by $r$}
    $cur \leftarrow r_I$                                    {Current blueprint}
reconf($r$)
    $target \leftarrow$ la-propose($r$)                     {See Section 2}
    $L \leftarrow L \cup \{target\}$
    $fr \leftarrow cur$ ; $s \leftarrow s_0$
    repeat
        $target \leftarrow \max(L)$                         {Maximum wrt. $\sqsubseteq$; $target$ for reconfiguration}
        $C[fr].writeNext(target)$                           {Record: $target$ was learned}
        $s_r \leftarrow C[fr].readS()$
        if $s.ts < s_r.ts$ then $s \leftarrow s_r$          {Remember most up-to-date register state}
        $L \leftarrow L \cup C[fr].readNext()$              {Check for learned blueprints}
        if $\exists r \in L : fr \sqsubseteq r$ then
            $fr \leftarrow \min(\{r \in L | fr \sqsubseteq r\})$   {Minimum wrt. $\sqsubseteq$}
        else break
    $C[target].writeS(s)$                                   {Transfer state to $target$}
    $C[target].rb.broadcast(\langle \textsc{Chosen}, target \rangle)$   {Inform about new configuration}
    return $target$
```

Algorithm 3 Get and Set Register State

```
get()
    $gt \leftarrow cur$ ; $s \leftarrow s_0$
    repeat
        $s_r \leftarrow C[gt].readS()$
        if $s.ts < s_r.ts$ then $s \leftarrow s_r$
        $L \leftarrow L \cup C[gt].readNext()$
        if $\exists r \in L : gt \sqsubseteq r$ then
            $gt \leftarrow \min(\{r \in L | gt \sqsubset r\})$
        else return $s$
```

```
set($s$)      {$s$ a (timestamp,value) pair}
    $st \leftarrow cur$
    repeat
        $C[st].writeS(s)$
        $L \leftarrow L \cup C[st].readNext()$
        if $\exists r \in L : st \sqsubseteq r$ then
            $st \leftarrow \min(\{r \in L | st \sqsubset r\})$
        else return
```

Algorithm 4 Atomic read/write at $\mathbf{p}_i$

```
read()
    $s \leftarrow$ get()
    set($s$)
    return $s.v$
```

```
write($v$)
    $(v', t') \leftarrow$ get()
    $t \leftarrow (t'.n + 1, \mathbf{p}_i.id)$
    $s \leftarrow (v, t)$
    set($s$)
```

Algorithm 5 Processing $\langle \textsc{Chosen} \rangle$

```
on $C.rb.deliver(\langle \textsc{Chosen}, target \rangle)$
with $cur \sqsubset target$
    $L \leftarrow L \cup \{target\}$
    $cur \leftarrow target$
    for $r \in L : r \sqsubset target$ do
        abort any method on $C[r]$
        $L \leftarrow L \backslash \{r\}$
```

5 Related Work

Previous work on reconfiguration of registers mainly use either the consensus-based approach or DirectCombine, as introduced in Section 1. Early work [16,17] assumed reconfigurations were issued by a single process. Thus avoiding the problem of concurrent reconfigurations, but failure of this process prevents further reconfigurations.

Several works use the consensus-based approach to handle concurrent reconfigurations. They either implement consensus [18,19], or assume an external, replicated configuration manager [3], [20]. All these systems establish a sequence of configurations. Since consensus is impossible in the face of asynchrony [21], these systems require additional assumptions, such as a failure detector or eventual synchrony. To guarantee liveness they assume, as we do, that an old configuration remains available until a newer configuration has started.

In [22] a group communication system is used to implement reconfiguration of an atomic register. This approach is similar to the consensus-based approach.

To our knowledge, DirectCombine has only been used in a few systems [4], [12,13]. These systems do not establish a sequence of configurations. Instead, processes can be added to or removed from the service at any time. To guarantee liveness they assume that only a bounded fraction of processes is removed concurrently [4], [12], or during a specific time interval [13]. Different from our work, [13] assumes an infinite sequence of reconfigurations. They also show this is impossible in an asynchronous system.

A replicated state machine (RSM) [23] is a general approach to replicate a service. An RSM can be used to implement atomic storage, where read and write operations are chosen using consensus. Consensus-based reconfiguration of an RSM was proposed in both [23,24] and has also been deployed in production systems [25]. In our previous work, we showed that an RSM can be reconfigured without relying on consensus [26]. In retrospect this work can be viewed as an application of SmartMerge. It uses a trivial combination function, that always chooses the configuration with the highest timestamp.

6 Conclusion

We presented an atomic register that uses a novel approach to combine concurrently issued reconfigurations in an asynchronous system. Our approach allows reconfigurations to specify a policy, that determines how to form a service configuration from the available processes. Different policies are aptly combined by a merge function.

References

1. Reiss, C., Tumanov, A., Ganger, G.R., Katz, R.H., Kozuch, M.A.: Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In: SOCC. (2012)

2. Ardekani, M.S., Terry, D.B.: A self-configurable geo-replicated cloud storage system. In: OSDI 2014
3. Gilbert, S., Lynch, N., Shvartsman, A.: Rambo: a robust, reconfigurable atomic memory service for dynamic networks. Distr. Comp. **23**(4) (2010)
4. Aguilera, M.K., Keidar, I., Malkhi, D., Shraer, A.: Dynamic atomic storage without consensus. J. ACM **58**(2) (2011) 7
5. Attiya, H., Bar-Noy, A., Dolev, D.: Sharing memory robustly in message-passing systems. J. ACM **42**(1) (January 1995) 124–142
6. Leners, J.B., Wu, H., Hung, W.L., Aguilera, M.K., Walfish, M.: Detecting failures in distributed systems with the falcon spy network. In: SOSP 2011
7. Leners, J.B., Gupta, T., Aguilera, M.K., Walfish, M.: Improving availability in distributed systems with failure informers. In: OSDI 2013
8. Vickers, S.: Topology Via Logic. Cambridge University Press (1989)
9. Faleiro, J.M., Rajamani, S., Rajan, K., Ramalingam, G., Vaswani, K.: Generalized lattice agreement. In: PODC 2012
10. Herlihy, M.P., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3) (July 1990) 463–492
11. Attiya, H., Herlihy, M., Rachman, O.: Atomic snapshots using lattice agreement. Distrib. Comput. **8**(3) (March 1995) 121–132
12. Shraer, A., Martin, J.P., Malkhi, D., Keidar, I.: Data-centric reconfiguration with network-attached disks. In: LADIS 2010
13. Baldoni, R., Bonomi, S., Kermarrec, A.M., Raynal, M.: Implementing a register in a dynamic distributed system. In: ICDCS 2009
14. Shao, C., Welch, J.L., Pierce, E., Lee, H.: Multi-writer consistency conditions for the shared memory objects. In: DISC 2003
15. Cachin, C., Guerraoui, R., Rodrigues, L.: Introduction to Reliable and Secure Distributed Programming. 2nd edn. Springer Publishing Company (2011)
16. Lynch, N.A., Shvartsman, A.A.: Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In: FTCS 1997
17. Englert, B., Shvartsman, A.A.: Graceful quorum reconfiguration in a robust emulation of shared memory. In: ICDCS 2000
18. Rodrigues, R., Liskov, B., Chen, K., Liskov, M., Schultz, D.: Automatic reconfiguration for large-scale reliable storage systems. IEEE Trans. Dependable Secur. Comput. **9**(2) (March 2012) 145–158
19. Chockler, G., Gilbert, S., Gramoli, V., Musial, P.M., Shvartsman, A.A.: Reconfigurable distributed storage for dynamic networks. Journal of Parallel and Distributed Computing **69**(1) (2009) 100 – 116
20. Martin, J.P., Alvisi, L.: A framework for dynamic byzantine storage. In: DSN 2004
21. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. J. ACM **32**(2) (April 1985) 374–382
22. Prisco, R.D., Fekete, A., Lynch, N.A., Shvartsman, A.A.: A dynamic primary configuration group communication service. In: DISC. (1999)
23. Schneider, F.B.: Implementing fault-tolerant services using the state machine approach: A tutorial. ACM Comput. Surv. **22**(4) (December 1990) 299–319
24. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. **16**(2) (May 1998) 133–169
25. Shraer, A., Reed, B., Malkhi, D., Junqueira, F.: Dynamic reconfiguration of primary/backup clusters. In: USENIX ATC. (2012)
26. Jehl, L., Meling, H.: Asynchronous Reconfiguration for Paxos State Machines. In: ICDCN 2014