The Case for Reconfiguration without Consensus: Comparing Algorithms for Atomic Storage

Leander Jehl[1] and Hein Meling[2]

1 **University of Stavanger, Stavanger, Norway**
`leander.jehl@uis.no`
2 **University of Stavanger, Stavanger, Norway**
`hein.meling@uis.no`

Abstract

We compare different algorithms for reconfigurable atomic storage in the data-centric model. We present the first experimental evaluation of two recently proposed algorithms for reconfiguration without consensus and compare them to established algorithms for reconfiguration both with and without consensus.

Our evaluation reveals that the new algorithms offer a significant improvement in terms of latency and overhead for reconfiguration without consensus. Our evaluation also shows that reconfiguration without consensus, can obtain similar results to that of consensus-based reconfiguration, which relies on a stable leader. Moreover, the new algorithms also substantially reduces the overhead compared to consensus-based reconfiguration without a leader.

While our analysis confirms our intuition that batching reconfiguration requests serves to reduce the overhead of reconfigurations, our evaluation also shows that it is equally important to separate reconfigurations from read and write operations. Specifically, we found that using read and write operations to assist in completing concurrent reconfigurations is in fact detrimental to the reconfiguration performance.

1998 ACM Subject Classification C.3.4 Distributed Systems, C.4 Performance of Systems

Keywords and phrases atomic storage, reconfiguration, data-centric model

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2016.31

1 Introduction

Cloud computing facilities offers an abundance of compute resources located in data centers around the globe. These data centers can host a wide range of services with different requirements and characteristics. For example, some services require replication for fault-tolerance. However, managing these data centers can be challenging, and often administrators will need to update both the composition of machines in the data center and the composition of replicas running a service. This is necessary to replace failed components, upgrade machine hardware, and adapt to changes in the service load. In practice, such reconfiguration operations are relatively frequent, as is evident from the traces of a Google data center [18].

To support reconfiguration, one of the main challenges faced by data center operators is to ensure consistency when multiple users submit concurrent reconfiguration requests. Moreover, a multitude of components may be monitoring software and hardware failures, upgrades, and the load of queries and updates to the replicas. Acting upon this information, reconfiguration requests may be issued autonomously, without human intervention [2]. We 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.

Rambo [12] was the first system to address reconfiguration of atomic storage. They used consensus to decide on reconfigurations. In [1] it was shown that such reconfiguration is possible in an asynchronous system, without having to solve consensus. However, experimental results [20] have shown that such reconfigurations add a significant overhead to concurrent read and write operations compared to reconfigurations using consensus. Recently however, two new approaches for reconfiguration without consensus have been proposed [14, 11]. These rely on lattice agreement (LA) [8] or a speculating snapshot algorithm (SpSn) [11] to reduce the worst case communication complexity of reconfigurations and operations concurrent with reconfigurations. While these two approaches appear to be superior in theory, understanding their behavior in different deployment scenarios is not obvious, since no experimental evaluation has been done.

Our main contribution in this paper is an extensive experimental evaluation of several algorithm variants of reconfigurable storage, SmartMerge-Store (SM-Store) [14], SpSn-Store [11], DynaStore [1], and Rambo [12]. SM-Store and SpSn-Store implement the novel approaches using lattice agreement and speculating snapshot, respectively.

Further, these reconfiguration algorithms are often presented in different models and formalisms. This has made it difficult to comprehend and compare the different algorithms.

Hence, our second contribution in this paper is a presentation of these algorithms, based on a common template aimed at highlighting their most relevant differences and similarities.

We implemented the algorithms in the data-centric model, in which processes are strictly separated into client and server roles, prohibiting servers from initiating communication. Section 2 describes this model in detail and motivate our choice.

In a local area network, our evaluation shows that, compared to DynaStore, the new algorithms' ability to batch reconfiguration requests can significantly reduce the overhead imposed on concurrent read operations. The new algorithms also have lower overhead than Rambo, if run without a stable leader, and similar overhead to leader-based Rambo.

Our evaluation indicates that treating read and write operations separately from reconfigurations is an important design principle. Different from SM-Store and Rambo, the SpSn-Store and DynaStore algorithms force read and write operations to help [4] in completing concurrent reconfigurations. However, we found that this actually increases the overhead that reconfigurations impose on read and write operations.

We implemented an optimization from Rambo, that allows a client to reuse a server's reply in the context of different configurations. We call this *single contact mode*. Our results show that this optimization can significantly reduce the overhead for read operations.

We also evaluated the algorithms for the case where clients and servers reside in different data centers across the globe. In this scenario, we found that the impact of batching reconfigurations is less pronounced, and in some cases DynaStore actually performs better than the new algorithms.

2 Why the Data-Centric Model?

The different algorithms in our study have been proposed and studied in different models. SpSn-Store [11] was presented in the data-centric model, while Rambo [12], SM-Store [14] and DynaStore [1] were all presented in the process-centric model. However, to the best of our knowledge, the only evaluation of reconfigurable storage without consensus was done in the data-centric model [20]. This section presents the two models, and motivate our choice of the data-centric model.

In the process-centric model, the processes invoking operations, e.g. reading and writing, also maintain the stored state and can respond to requests from other processes.

In the data-centric model, we distinguish between client and server processes. Clients perform operations, while servers respond to client requests and maintain state. Clients and servers can, but need not be located on the same machines. This model restricts communication, in that servers only respond to client requests. That is, servers cannot initiate communication with clients and two servers do not communicate directly.

The data-centric model is generally considered to be more scalable [20, 6] since it reduces bottlenecks and avoids all-to-all message patterns common in the process-centric model. Moreover, the set of clients can easily be changed without changing the set of servers.

It is not clear how to compare latencies and throughput achieved in the different models. Furthermore, in DynaStore and SM-Store if a process is removed while performing an operation, this operation may never complete. This makes it difficult to measure the latencies of operations concurrent with reconfigurations. We therefore implemented all algorithms in the data-centric model. The algorithms in our study are all a good match for this model, since their interactions mostly follow the request-reply pattern between clients and servers. With this choice, our results are also directly comparable with the results from [20]. This is relevant, since part of our motivation is to test whether the new protocols for reconfiguration without consensus (SpSn-Store and SM-Store), also introduce the significant overhead to *read* and *write* operations, as observed in [20].

However, in the data-centric model an idle client cannot be informed by the servers, when one or more reconfigurations together replace all of the servers with new ones. This problem arises in many reconfiguration algorithms and is typically solved using a resource discovery service [20, 17, 21]. Even in systems where idle clients are notified, a resource discovery service is still needed to allow new clients to join. Since our evaluation is focused on the *read* and *write* performance during reconfiguration, we refer to these other works for solutions to the discovery problem.

To use a **single leader** to propose reconfigurations is an easy way to avoid concurrent reconfigurations, and thus the main difficulty of reconfiguration. This is especially relevant for Rambo, which uses consensus to choose one of several concurrent reconfigurations, because an established leader can skip the first phase of consensus [16].

To ensure a fair comparison for Rambo's consensus-based algorithm, we have implemented two variants of this algorithm; one where clients forward reconfiguration requests to a leader, and one where every process believes itself to be the leader. We refer to the leader variant as L-Rambo. Note that L-Rambo does not entirely comply with the data-centric model. A leader must both receive requests from other clients, and perform operations on the servers. It is therefore both a client and a server. Accordingly, we found that introducing the additional role of a leader significantly increased the complexity of implementing and deploying this variant. While this assessment is clearly subjective, for us it validated the claim, that the data-centric model promotes simplicity. The other algorithms, besides Rambo, could also benefit from a leader batching reconfigurations. However, we believe it is more interesting to compare the leaderless algorithms with L-Rambo.

3 Reconfigurable Storage Interface

The algorithms in our study provide three operations, *read*, *write*, and *reconf*. The *write* operation stores a single value and the *read* operation returns the last value that was written. The *reconf* operation is used to change the set of servers and is discussed below. In all algorithms in our study the *read* and *write* operations fulfill atomic [15] semantics. Thus,

even if several operations are executed concurrently, they appear as if all operations where executed sequentially.

The first algorithm to implement atomic *read* and *write* objects in an asynchronous system, that is subject to failures, was the ABD algorithm [3]. All algorithms in our study are based on this work. In the ABD algorithm, values are always stored together with a logical timestamp. We refer to such a (timestamp, value) pair as a timestamped value, and say that one timestamped value is greater than another, if it has a higher timestamp. The ABD algorithm assumes a fixed set of servers. Values are read from and written to a majority of these servers.

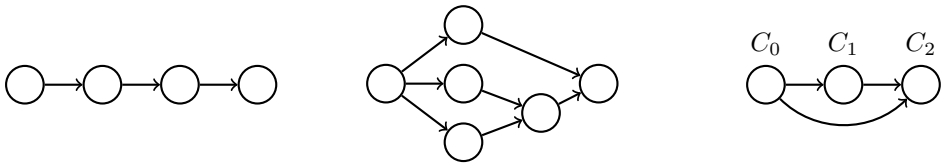
In ABD, both *read* and *write* operations proceed in similar manner, performing first a query, followed by a propagation phase. During the query phase a client collects timestamped values from a majority of the servers. Among the collected values, the client determines the one with the highest timestamp. In a *read* operation, before returning this value, the client propagates the value and timestamp back to a majority, to ensure that successive operations will also read this value. A client performing a *write* operation, on the other hand, uses the highest timestamp found in the query phase along with its process identifier, to create a higher, unique timestamp, and propagates its own value, together with the new timestamp to a majority of servers.

Thus, *read* and *write* operations only differ in the value and timestamp they propagate, not in the actual set of messages that need to be sent and received. This is also true for the reconfigurable algorithms in this study. The ABD algorithm can be optimized to allow some reads to return after the query phase [7]. The applicability of such optimizations depends on the workload, since usually only reads that are not concurrent with a *write* operation can be optimized. Similarly, a regular *read* operation only performs the query phase of the atomic *read* operation, but provides weaker consistency [19]. In our study, we have implemented regular reads, to enable comparison with the most impactful optimization.

There also exist optimizations for managing large objects, e.g. by separating stored objects from metadata [9]. Using the algorithm from [9], the reconfiguration of servers storing the actual data is easy and its performance mainly depends on the size of the state. The algorithms in this study could be used for reconfiguration of metadata servers.

All algorithms organize the servers into configurations. A configuration is a set of servers. In SM-Store and Rambo, a configuration also includes a read-write quorum system, while DynaStore and SpSn-Store assumes that a majority quorum is used. In our implementations of SM-Store and Rambo a write-quorum consists of a majority of the processes in a configuration, while a read-quorum consists of at least half the processes in the configuration. This way, the quorum systems used in SM-Store and Rambo provide the same fault tolerance as the majority quorums used in DynaStore and SpSn-Store, while SM-Store and Rambo may still benefit from their ability to use a more flexible quorum system.

Besides *read* and *write*, all algorithms in our study allow a *reconf* operation to change the configuration. In DynaStore and SpSn-Store, this operation takes a set of tuples $(s_i, +)$ or $(s_j, -)$ as input, where $(s_i, +)$ signals that server s_i should be added, while $(s_j, -)$ removes s_j. We call a set $\{(s_i, +), (s_j, -), ...\}$ a *change*. DynaStore and SpSn-Store assume an initial configuration C_0. Every other configuration C_l is identified by a change ch_l, such that C_l results from applying ch_l to C_0. We write $C_l = ch_l(C_0)$. To apply an additional change ch to C_l we simply add ch to ch_l: $ch(C_l) = ch \cup ch_l(C_0)$. In these algorithms, after a *reconf*$(\{(s_1, +), (s_2, -)\})$ operation returns, server s_2 is no longer part of the current configuration, and s_1 is part of the configuration, unless it was explicitly removed by another reconfiguration.



(a) Rambo uses consensus to choose a successor for every configuration.

(b) In **DynaStore**, configurations can have multiple successors.

(c) Using **LA** or **SpSn**, we can ensure that configurations are ordered.

Figure 1 Directed acyclic graphs of successor configurations. Circles are configurations and arrows are established successors.

In the same way, SM-Store's *reconf* operation takes a change as argument. Furthermore, the *reconf* operation takes a policy that can be used to specify additional changes, e.g. to the quorum system. However in the context of this paper, we are interested in comparing the performance of reconfigurations that can be specified in all algorithms, and thus we always use an empty policy.

Rambo's reconfiguration interface differs from that of the other algorithms, where a reconfiguration only returns after its change has been applied. That is, a *rambo-reconf* proposes a new configuration C to a consensus instance in the current configuration *cur*. Only if C is chosen by this consensus, C will become the new current configuration. We therefore create a wrapper for *rambo-reconf* that has the same semantics as the other *reconf* operations. This *reconf* operation receives a change as argument, then applies this change to the current configuration *cur*, and invokes *rambo-reconf*(*change*(*cur*)). If some configuration C' was chosen by consensus that does not include the change proposed by our reconfiguration, we apply our change again, invoking *rambo-reconf*(*change*(C')). Thus a reconfiguring client may need to invoke several *rambo-reconf* operations before its change is applied and it can return from the *reconf* operation.

4 A Common Template for Reconfiguration Algorithms

We now present a common template for the different reconfiguration algorithms that we evaluate. We slightly simplified the algorithms for this presentation, to better highlight relevant similarities and differences. In our implementation we follow the original version of the algorithms.

4.1 The Graph of Successor Configurations

To apply a change *ch* to the current configuration C, a client must register this change with the servers in configuration C. Registered changes create a directed acyclic graph (DAG) of configurations, where an arc connects C to $ch(C)$, if the change *ch* was registered with C. In this case we say that $ch(C)$ is a successor of C. Figure 1 shows some examples.

Registering a change might fail. For example, in Rambo only a single change is chosen by a configuration (Figure 1a). Thus a reconfiguring client must traverse the graph, until it can record its changes. In the other algorithms a configuration may have multiple successors (Figures 1b and 1c). In this case, a client must traverse the graph to ensure that the new configuration is a direct or indirect successor of every other configuration in the graph.

Our common template for the reconfiguration algorithms, depicting such a graph traversal, is shown in Figure 2. In every visited configuration, the client tries to establish a new

successor configuration that includes the requested *change* and collects information on existing successors. The function ESTABLISH&COLLECTSUCCS(*change*) on Line 9 of the template is implemented differently by each algorithm, as we describe below. The client then updates the DAG with the collected successors on Line 12. The changes realized in these successors are also added to the proposed *change* on Line 13. This ensures that concurrent reconfigurations do not cancel each other, but eventually all reach the same configuration. The client collects the state from each visited configuration, and transfers the most up-to-date state to the new configuration (Lines 10–15). On Line 16 the client starts the new configuration. Future reconfigurations may then begin their traversal from this configuration and old configurations can be discarded.

We had to adjust how configurations are started, to fit with the data-centric model. The original versions of Rambo, SM-Store, and DynaStore use an all-to-all broadcast to inform all clients that a new configuration was started. This violates the assumptions of the data-centric model, which disallows broadcasting to all clients. In our implementation, the reconfiguring client performs *cur*.start() by informing the servers in *cur* that this configuration was started. When replying to other clients, the servers include this information, allowing also other clients to discard old configurations. This approach was also used in [20] to adapt start() to the data-centric model.

Algorithm 1 shows two common primitives that we use to describe the different algorithms. We assume that every server s_i in a configuration *cur* stores a set of proposed changes Ch_i. Each of these changes corresponds to a successor configuration *change*(*cur*). The *cur*.ADDCHANGES($\{ch_1, ch_2, ...\}$) primitive tells every server s_i in *cur* to add the changes ch_1 and ch_2 to its change set Ch_i, thus adding two new successor configurations. ADDCHANGES only returns after a majority of the servers have applied this update. Note that each of the changes ch_1 or ch_2 may include several additions and removals. However, only in DynaStore is ADDCHANGES actually called with a set of changes. Similarly, *cur*.GETCHANGES() reads the Ch_i variables at a majority of the processes in *cur* and returns the union of these sets.

For **Rambo**, the implementation of ESTABLISH&COLLECTSUCCS(*change*) is shown in Algorithm 2. The reconfiguring client proposes its *change* to the consensus instance in configuration *cur* and learns a possibly different set of changes *ch* that has been chosen by consensus. We use the Paxos consensus algorithm [16], however, we do not use all-to-all learn messages, since they do not comply with the data-centric model. Instead we only send learn messages to the client that proposed a value. After learning a new configuration from Paxos, the reconfiguring client then informs the servers in *cur*, that *ch* was chosen, invoking *cur*.ADDCHANGES(*ch*). It is not necessary to collect successors, since no other change than *ch* can be chosen by consensus.

We note that there exists an optimized variant of Rambo, called RDS [5]. In RDS the servers in an old configuration forward their state directly to all servers in the new configuration. This reduces the number of message delays necessary for a reconfiguration. We have not implemented this optimization, since it relies on an all-to-all message exchange among servers and is thus not applicable to the data-centric model.

Since Rambo only chooses a single successor for every configuration, the graph of configurations has a single path, as shown in Figure 1a. In L-Rambo, where a leader performs reconfigurations on behalf of other clients, this leader can combine the changes proposed by different clients in a single configuration and propose this to the consensus algorithm. We refer to this process as batching.

Without consensus it is not possible to choose a single successor. Thus in **DynaStore**, multiple successors can be established for one configuration. These successors must eventually

Template for reconfiguration.

```
1:  State :
2:      cur                                              ▷ current configuration
3:  reconf(change)
4:      allchanges := change                             ▷ set to collect all changes applied in this reconf
5:      DAG := cur                                       ▷ graph with single node
6:      S := {}                                          ▷ set of timestamped values
7:      while allchanges(cur) ≠ cur do
8:          cur ← next ∈ DAG in topological order
9:          Ch ← cur.ESTABLISH&COLLECTSUCCS(allchanges)  ▷ try to establish successor, collect successors
10:         S ← S ∪ cur.collectState()                   ▷ collect timestamped values from majority
11:         for ch ∈ Ch do
12:             DAG.add(cur → ch(cur))                   ▷ add successor arc to graph
13:             allchanges ← allchanges ∪ ch             ▷ combine changes
14:     v := max_{v.ts}(S)                               ▷ find value with highest timestamp
15:     cur.updateState(v)                               ▷ update timestamped value at majority
16:     cur.start()                                      ▷ Inform servers in cur that cur is started
```

Alg. 1 Auxiliary functions (RPCs to access state at servers)

```
State at every server s_i:
    Ch_i := {}                                   ▷ set of changes
function cur.ADDCHANGES({ch_1, ch_2, ...})
    for all s_i ∈ cur invoke in parallel
        at s_i do: Ch_i ← Ch_i ∪ {ch_1, ch_2, ...}   ▷ remote
    wait until assignment completed at majority in cur
function cur.GETCHANGES
    for all s_i ∈ cur invoke in parallel
        Ch_i := s_i.read(Ch_i)                   ▷ read from remote
    wait until s_i.read(Ch_i) completed at majority in cur
    return ⋃ Ch_i                                ▷ only those with completed read
```

Alg. 2 Rambo: traversal

```
1:  cur.ESTABLISH&COLLECTSUCCS(change)
2:      ch ← cur.consensus.propose(change)   ▷ 1 to ∞ round trips
3:      cur.ADDCHANGES({ch})                 ▷ 1 round trip
4:      return {ch}
```

Alg. 3 SpSn-Store: traversal

```
1:  cur.ESTABLISH&COLLECTSUCCS(change)
2:      return cur.SpSn(change)              ▷ 2 to 2r round trips
```

Alg. 4 SM-Store: traversal

```
1:  cur.ESTABLISH&COLLECTSUCCS(change)
2:      ch_LA ← cur.LA(change)        } 1 to r
3:      Ch ← cur.GETCHANGES()         }
4:      if ∃ch ∈ Ch then              } 1 round trip
5:          cur.ADDCHANGES({ch})      }
6:      else                          }
7:          cur.ADDCHANGES({ch_LA})   }
8:      return cur.GETCHANGES()       }
```

Alg. 5 DynaStore: traversal

```
1:  cur.ESTABLISH&COLLECTSUCCS(change)
2:      ch ← {at some s_i ∈ cur do:        }
3:          if Ch_i == { } then            } 1 round trip
4:              Ch_i ← {change}            }
5:          return some ch ∈ Ch_i          }
6:      }                                  ▷ end remote procedure
7:      cur.ADDCHANGES({ch})               } 1 round trip
8:      Ch ← cur.GETCHANGES()              }
9:      cur.ADDCHANGES(Ch)                 } 1 round trip
10:     Ch ← cur.GETCHANGES()              }
11:     cur.ADDCHANGES(Ch)   ▷ omitted if identical to Line 9
12:     return Ch
```

Figure 2 Pseudocode for reconfiguration. Client code and remote procedures invoked on servers.

be merged into a single configuration (see Figure 1b). Algorithm 5 shows how successors are established (Lines 2–7) and collected (Lines 8–11). Lines 2–6 represent a best effort approach to limit the number of successors. This was not part of the original DynaStore algorithm, but introduced in [20]. Here the reconfiguring client contacts a single server. If this server already knows of a different successor, the client will not establish a new successor. In [20] the client invokes this remote operation concurrently on a majority of the servers, but waits for only one of them to return. In our implementation, we only perform this operation on the server with lowest ID. Only if this server fails to reply, do we perform the operation on a majority of the servers. Further, if multiple clients try to register a change with a configuration C, they all try to contact the same server for Lines 2–6. Thus in the normal case, only a single successor will be established.

In DynaStore a client performs two calls of GETCHANGES to collect successors. The client also calls ADDCHANGES twice, to ensure that other clients will collect the same successors.

We refer the reader to [1, 20] for a more detailed explanation of this mechanism. In our implementation we omit the call to addChanges on Line 11 if its argument is the same as the one already used on Line 9.

Algorithm 4 shows the establishing and collecting of successors for **SM-Store**. Before calling addChanges() and establishing a new successor, a reconfiguring client proposes its changes to lattice agreement. We implement the lattice agreement algorithm from [8]. In this algorithm the client repeatedly writes and collects proposals from a majority of processes. The client adjusts its proposal, until it includes all proposals made by other clients. The change proposed to lattice agreement is included in the returned change. Further, for two changes ch_1 and ch_2 returned from lattice agreement, either ch_1 is part of ch_2 or vice versa. Thus, even though a configuration in SM-Store can have several successors, these will be ordered, e.g. all changes realized in configuration C_1 are also part of configuration C_2 (see Figure 1c). If several clients invoke lattice agreement concurrently it is likely that they all learn the same change, combining all proposed changes. Thus, the different reconfigurations will only add a single configuration to the graph. We say that the reconfigurations are batched. Pseudocode for lattice agreement can be found in [13].

To ensure that not only the successors to one configuration, but all successors are ordered, it is important that a reconfiguring client solves lattice agreement in a configuration that does not yet have a successor. Therefore, a reconfiguring client invokes getChanges on Line 3 and only if this returns an empty set, will the client use the value returned from lattice agreement to establish a new successor on Line 7. Otherwise the client enforces an existing successor (Line 5). The collection of successors is done with a simple call to getChanges on Line 8.

The **SpSn-Store** uses a speculating snapshot algorithm to both establish and collect successors. Speculating snapshot is similar to lattice agreement used in SM-Store, in that it can combine concurrently proposed changes and all established successors are ordered. Thus, the resulting graph of configurations becomes similar to SM-Store (see Figure 1c). Like SM-Store, concurrently proposed changes can also be batched using speculating snapshot. However the actual algorithm for speculating snapshot, given in [11] is quite different from lattice agreement.

A client invoking speculating snapshot performs several rounds of message exchanges, where each round has two phases. In the first phase a client disseminates its own changes and collects changes proposed by others in the same round. If no other changes are proposed, the client commits its proposal in the second phase. A committed value represents a successor configuration. If other changes have been proposed, the client disseminates all changes it has collected in the second phase. The client also collects values committed by other processes. Finally, the client starts a new round, proposing the combination of all changes observed in previous rounds. Thus in every round, at most one value is committed.

In our implementation, a client contacts a majority of the servers once for every round and phase, accessing different local variables depending on the round and phase. We refer the reader to [11] for a more thorough explanation of this algorithm. Pseudocode for the speculating snapshot algorithm can also be found in [13].

For our implementation we optimized the message pattern above using the following principle: If a call to *cur*.addChanges is always followed by *cur*.getChanges(), we simply include the local variables Ch_i in the reply returned by *cur*.addChanges. Thus, the two calls can be implemented using a single message round trip. We included braces in our pseudocode above, to show which calls are combined into one round trip. Additionally, we include the timestamped value stored at the servers in one of the message exchanges

Table 1 Differences between the studied algorithms.

	Rambo	L-Rambo	DynaStore	SpSn-Store	SM-Store
can batch concurrent reconfigurations	no	yes	no	yes	yes
round trips for establish and collect	3 to ∞	2	3 to 4	2 to $2r$	2 to $r+1$
read and *write* establish successors	no	no	yes	yes	no

performed in ESTABLISH&COLLECTSUCCS. Thus, no additional messages are necessary to collect these values on Line 10 of our template.

4.2 The Cost of a Traversal

We now perform a brief analysis of the cost of a traversal. This cost is related to the size of the successor graph that must be traversed, and the cost of establishing and collecting successor relations. We summarize this discussion in Table 1.

In Rambo, r reconfigurations representing different changes will add r configurations to the graph, since every reconfiguration creates one successor. In L-Rambo, the leader can batch these reconfigurations into fewer configurations. A stable leader can solve consensus and inform the servers about the outcome in two round-trips, thus establishing a successor. A new leader requires an additional round-trip. However, in an asynchronous system, multiple leaders may compete indefinitely for leadership and never achieve consensus [10]. Note also that a reconfiguring client may have to participate in several consensus instances, until its proposed change is chosen.

In DynaStore, r reconfigurations result in at least r new configurations. If the reconfigurations are combined in different orders by different clients, this can theoretically result in a successor graph with $2^{r-1}+r$ configurations. To traverse a single configuration normally requires only three round trips. The first of these only needs to contact a single server, not a majority. As explained above, this is because we obtain the values needed by GETCHANGES on Lines 8, 10 of Algorithm 5 from the replies of ADDCHANGES on Lines 7, 9. We omit the call to ADDCHANGES on Line 11, if its argument is the same one used on Line 9.

In SM-Store, r reconfigurations create at most r configurations, but concurrent reconfigurations may also be batched, resulting in fewer configurations. A single client requires only a single round trip to solve lattice agreement and another round trip to establish and collect successors. If r clients invoke concurrent reconfigurations, they may require r round trips to solve lattice agreement.

For SpSn-Store, r reconfigurations create at most r new configurations, possibly less. A single client can solve speculating snapshot in only two round trips, but r clients, proposing different changes concurrently, may require up to r rounds and therefore $2r$ round trips.

4.3 Read and Write Operations

In a reconfigurable storage, clients must check for successor configurations both after the query and dissemination phase of every *read* and *write* operation. If no successor configurations are found, a *read* or *write* operation precedes as in a stable system.

There are two approaches to handle a successor found during a *read* or *write* operation. In SM-Store and Rambo, a *read* or *write* operation simply traverses the graph of successor

configurations, and executing the query phase in all these configurations and similarly for the dissemination phase.

In SpSn-Store and DynaStore, when a *read* or *write* operation finds a successor, they start a reconfiguration towards this successor configuration. A *read* operation then simply returns the state collected during the traversal (v on Line 15 of the template). A *write* writes its own value to the new configuration on Line 15, instead of the collected value.

Performing a reconfiguration is clearly more costly than simply reading from or writing to all configurations in the graph. However, by performing a reconfiguration, a client ensures that any edge traversed in one operation, will not have to be traversed again by successive operations from this client. This may happen with the first approach, especially if a reconfiguring client fails while performing a reconfiguration.

5 Implementation

We have implemented all algorithms in Go 1.5 (`http://golang.org`). All algorithms implement a single register. Servers keep the algorithm state in memory and clients can only read or write the complete register. We build on a quorum RPC framework that clients use to communicate with servers. A quorum RPC sends a request to all servers in a configuration and returns after receiving replies from a quorum. Our clients always block on a quorum RPC. Our implmentation is available at `http://www.github.com/relab/smartmerge`.

Thrifty mode. Since the algorithms in our study are designed for an asynchronous system subject to failures, none of our RPCs actually require a reply from all servers in a configuration. We therefore designed a thrifty mode, where an RPC is only sent to a quorum of processes, and only after a timeout will the RPC be sent to all processes in the configuration. In our experiments, we configured this timeout to avoid resend in the absence of failures. Unless noted otherwise, all experiments are done in thrifty mode. In our implementation a client sends all thrifty RPCs to the same quorum. Different clients use different quorums to ensure that load is evenly distributed among servers.

Single contact mode. In Rambo, during the query or dissemination phase of a *read* or *write* operation, a client performs the same RPC on all configurations in the successor graph. A server's reply to this RPC in one configuration can also be used as reply in another configuration. This is possible because the servers in Rambo do not store or send information specific to a configuration. Instead, the servers send the largest timestamped value received in any configuration, and the whole successor graph, omitting only garbage collected configurations. We call this *single contact mode* (SCM) and we have implemented it for both Rambo and SM-Store. The reason SCM is also applicable to SM-Store, is that *read* and *write* operations in SM-Store are very similar to the ones in Rambo.

6 Evaluation

We evaluated the algorithms both in a local area (LAN) and wide area (WAN) setting. We evaluate both how quickly reconfigurations are applied, and the overhead these reconfigurations impose on concurrent operations.

While we use TCP for communication in all our experiments, we start servers and establish connections at startup, not during reconfigurations. This allows our evaluation to focus on the cost of running the specific reconfiguration algorithm. We believe this to be useful also

in practice, since it is possible to tell the clients to establish connections to a new server, before the reconfiguration to add this server, is actually performed.

Further, the clients in our experiments only perform *read* operations, not *write* operations, since atomic *read* operations only differ from *write*s in a small computation done locally at the client. If all values written and read have approximately the same size, then *read* and *write* operations differ neither in the kind of messages sent, nor their size.

Experiment setup. We evaluate the algorithms in a Gigabit LAN environment with machines running Linux 3.18.2. We use "small" machines with a 1.86 GHz Intel Core dual-core processor to run two servers each, and "large" machines with a 2.13 GHz Intel Xeon quad-core processor to run four clients each.

We start our experiment with an initial configuration of 8 servers each on a different "small" machine. We let 16 clients on four "large" machines continuously perform *read*s, with a payload of 4 kB. In absence of reconfigurations, these *read*s utilize the servers with more than 80 % of their maximal throughput. At one point during the experiment we start 1, 2, or 3 clients, each issuing a reconfiguration to replace one of the servers with another server, located on the same machine. Thus, every configuration actually retains the same number of servers, located on the same machines. The reconfiguring clients are located on another "large" machine. For L-Rambo we use another "large" machine to run the leader. In this setup, the leader of L-Rambo is rather over-provisioned. Initial experiments suggested, that using one of the servers in the initial configuration as leader increases reconfiguration latencies by 10-15 %, compared to the results reported below.

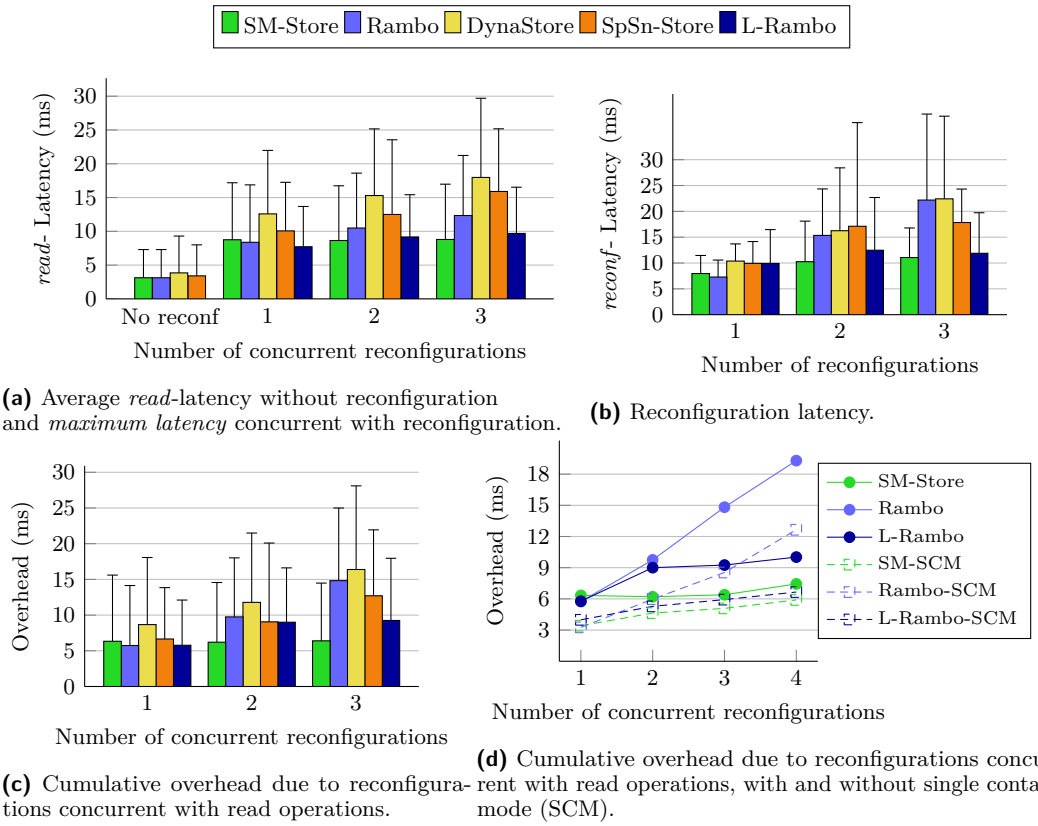
All LAN experiment are done using thrifty mode. We performed initial experiments to verify that this mode actually improves performance in all algorithms. Due to space constraints we do not report on these experiments.

Metrics. We measure both the time it takes to complete a reconfiguration and the overhead that this reconfiguration impose on concurrent read and write operations. The first measure is simply the latency of reconfiguration operations. To measure the overhead, we measure the latency of read and write operations, and label the latencies of those operations that contact several configurations. If the latency of an operation is labeled, and it is higher than the average of unlabeled latencies, we call this difference overhead.

However as we mentioned in Section 4.3, depending on how reads and writes handle successive configurations, a single reconfiguration can cause overhead to one or more operations from a client. We therefore use two metrics to evaluate the overhead. The *cumulative overhead* for a client is the total overhead that the client experienced in one run. The *maximum latency* is the maximum latency any operation from a single client experienced in one run.

While maximum latency is relevant to all clients, we believe that a small cumulative overhead is only relevant to clients that perform frequent operations.

Concurrent reconfigurations. Figure 3 shows results for handling 1-4 concurrent reconfigurations. Figure 3a shows the average read latency without reconfiguration and the **maximum read latency** a client experiences concurrent with one or more reconfigurations. The figure shows the average maximum latency and the 95th percentile for 16 clients in 40 runs. We observe that read latencies increase significantly for all algorithms during reconfiguration, but this overhead differs significantly for different algorithms. The different overhead can be explained by the characteristics we summarized in Table 1. Rambo, L-Rambo and SM-Store perform better than DynaStore and SpSn-Store since in these algorithms, *read* operations do



(a) Average *read*-latency without reconfiguration and *maximum latency* concurrent with reconfiguration.

(b) Reconfiguration latency.

(c) Cumulative overhead due to reconfigurations concurrent with read operations.

(d) Cumulative overhead due to reconfigurations concurrent with read operations, with and without single contact mode (SCM).

Figure 3 Processing a single batch of reconfigurations. Average and 95th percentile for 16 clients over 40 runs.

not establish successors, i.e. they do not complete concurrent reconfigurations. For two or more concurrent reconfigurations, SM-Store and L-Rambo perform better than Rambo, since multiple reconfiguration are batched. Thus, also fewer configurations have to be traversed by *read* operations. SpSn-Store also batches reconfigurations. but this seems to have little significance.

Figure 3b shows the actual **reconfiguration latencies**. We see again that latencies scale well for L-Rambo and SM-Store due to batching. On the other hand, latencies increase drastically for Rambo and DynaStore which do not make use of any batching.

Figure 3c shows the **cumulative overhead** due to reconfigurations. DynaStore experience the highest overhead. This is the combined effect of the lack of batching and *read* operations completing concurrent reconfigurations. The overhead of Rambo increases significantly, as more reconfigurations are invoked, due to the lack of batching. The overhead of Rambo actually grows faster than the *read* latencies for Rambo. That is because usually the initial configuration is already removed from the DAG, when the second or third configuration is added. Thus no single *read* operation needs to contact all configurations. SM-Store scales well since even three reconfigurations are batched into a single new configuration. In L-Rambo, only the second and third reconfiguration are batched. This is because we do not use any batching timeout, but instead propose the first reconfiguration immediately.

We now evaluate the **single contact mode**, where clients try to avoid contacting the same process in different configurations. We have implemented this mode for Rambo, L-Rambo, and SM-Store, as explained in Section 5. The experiment setup is the same as

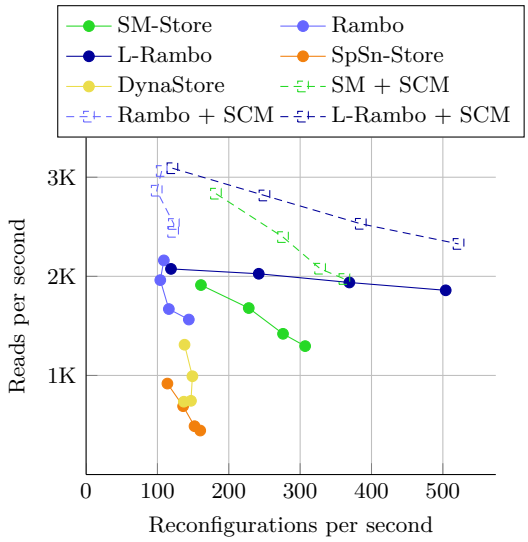


Figure 4 Read and reconfiguration throughput with 2, 4, 6, and 8 reconfiguring clients, each represented by a dot or square in the graphs. More clients increases reconfiguration throughput. Average over 20 runs, each with a 30 second duration.

in the previous experiment. Figure 3d shows that cumulative overhead, with and without single contact mode, in scenarios with 1, 2, 3, and 4 concurrent reconfigurations. We see that, while its effect on SM-Store is limited, single contact mode has a significant impact on L-Rambo, mitigating the difference between L-Rambo and SM-Store. For Rambo without leader, single contact mode partially mitigates the lack of batching. However, for a larger number of concurrent reconfigurations (e.g. 4), Rambo still experiences significantly larger overhead than the other variants.

We repeated the above experiments using **regular reads** that omit the dissemination phase from *read* operations. This experiment also serves as an estimate for optimizations that maintain atomic semantics, but omit the dissemination phase when possible. Such optimizations (e.g. RDS [5]) are equally applicable to Rambo, L-Rambo and SM-Store. Omitting the dissemination phase from *read* operations reduces normal case *read* latencies by 66% for SM-Store and Rambo and 49% for DynaStore and SpSn-Store. Maximum latencies during reconfiguration are also reduced by 40-50% in SM-Store, Rambo and L-Rambo. But *read* operations that complete concurrent reconfigurations (DynaStore and SpSn-Store) maintain the same latencies. Operations that complete concurrent reconfigurations must disseminate values to the new configuration. Thus the dissemination phase cannot be completely omitted in DynaStore and SpSn-Store.

Measurements on overhead, reconfiguration latency and the impact of single contact mode are qualitatively similar to the ones reported above.

Constant reconfiguration. We are also interested in the questions: what frequency of reconfigurations can the algorithms support, and how does a constant rate of reconfigurations impact *read* throughput?

Under constant reconfiguration, the algorithms cannot guarantee that operations complete. That is because reconfigurations might be adding configurations to the graph faster than a *read* operation can traverse this graph. However, our experiment shows that several algorithms can still maintain reasonable throughput.

We use 8 servers and 16 *read*-clients as in the previous experiment. We then start 2, 4, 6, and 8 clients that continuously replace different servers, switching back-and-forth between two servers located on the same machine.

Figure 4 plots the number of completed reconfigurations against the number of completed read operations per second. We see that Rambo, SM-Store, and especially L-Rambo maintain reasonable read throughput.

For most algorithms, additional reconfiguring clients result in more completed reconfigurations. DynaStore, SpSn-Store, and Rambo all complete between 100 and 160 reconfigurations per second. However, in Rambo significantly more *read* operations complete concurrent with these reconfigurations than in DynaStore and SpSn-Store. That is, because *read* operations in Rambo do not complete concurrent reconfigurations. In DynaStore, adding additional reconfiguring clients does not improve reconfiguration throughput but still reduces *read* throughput. That is because in DynaStore all reconfigurations and *read* operations that complete concurrent reconfigurations try to contact a single server to establish a successor configuration (see Section 4). In this experiment, this single server becomes a bottleneck.

Figure 4 also shows results for single contact mode. This mode significantly improves the read throughput but has little effect on the number of completed reconfigurations.

6.1 WAN experiments

In this section we present experiments performed in a wide area network (WAN).

We performed similar experiments to those reported above, using Amazon Web-Services micro instances running Ubuntu 14.04 in several data centers. We used a different instance for each client and server.

In our experiments we started with a configuration with 3 servers and 3 clients continuously performing *read* operations. A client and a server were located in each of Europe (Frankfurt), US West (N. California) and Asia (Tokyo). The *read* operations have a payload of 100 bytes. For L-Rambo we use an additional instance located in US West as leader.

We did not use thrifty mode in these experiments, because using this mode in a wide area network requires the client to carefully choose the servers with the lowest latency. This is especially difficult to determine for servers in new configurations where a client cannot rely on the latencies from previous requests.

We first evaluate the algorithms under **constant reconfiguration**. For this experiment we use three clients, located in each of the above mentioned data centers. During 30 runs each lasting 60 seconds, the clients constantly propose reconfigurations. Every reconfiguration proposes to replace the server located in the same data center as the reconfiguring client, with a server, located on another instance in the same data center. Figure 5 shows average latencies for these *reconf* operations and concurrent *read*s and the 95th percentile. Note that the two measurements must be seen in conjunction. For example, since reconfigurations in Rambo often take several seconds to complete, only a few reconfigurations actually complete in a run. Few reconfiguration only cause a small overhead to *read* operations.

The *reconf* latencies for Rambo, SM-Store are dominated by extreme spike latencies. SpSn-Store also experiences some extreme spike latencies, which may exceed the experiment duration (> 60 seconds). While one client experiences a spike latency, another client may preform many reconfigurations. Thus spike latencies form less than 5% of the observed latencies and have little effect on the 95th percentile, but cause the average to lie above this percentile.

The high spike latencies for reconfigurations all come from the reconfiguring client in Europe. In Rambo and SpSn-Store, in some runs, this client does not manage to successfully

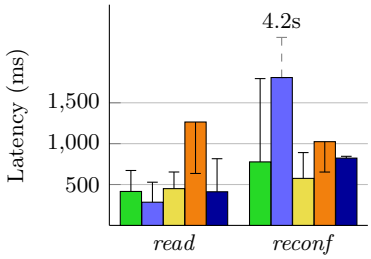
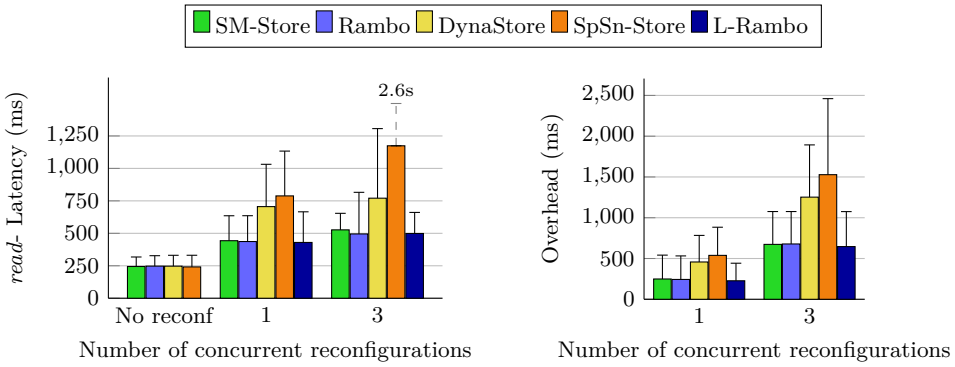


Figure 5 Latencies in WAN under constant reconfiguration. Average and 95th percentile for 30 runs lasting 60 seconds each.



(a) Read latency in WAN without, with 1 or with 3 concurrent reconfigurations.

(b) Overhead in WAN with 1 or 3 concurrent reconfiguration.

Figure 6 Latencies in WAN. Average and 95th percentile obtained from 30 runs.

apply its changes, before the end of the experiment. In SpSn-Store this also happened to the European client performing reads.

In SM-Store the reconfiguring client from Europe always manages to complete a request during the experiment, but requires up to 30 seconds to do so. We see that in SpSn-Store these spike latencies extend to *read* operations, since also *read* operations participate in the Speculating snapshot. In Rambo and SM-Store on the other hand, the spike latencies for reconfigurations have little impact on the *read* latencies.

Surprisingly, in this experiment DynaStore performs especially well, with the lowest average reconfiguration latency of all algorithms in our study, and an average read latency that is similar to the other algorithms. As described in Section 4, DynaStore uses one of the servers in the configuration to prevent multiple successors. In this experiment, this server is located in Europe, which gives an advantage to the reconfiguring client located in Europe. It is this client that experiences spike latencies in the other algorithms.

We also measured the latency and overhead of a **single batch of reconfigurations** in our wide area setting. In this experiment, we first let a single client propose a reconfiguration, replacing one server with a new one, located in the same data-center. We alternate both on the location of the client and, which server is reconfigured. We also performed an experiment, where all three reconfiguring clients, one in each data center, concurrently perform one reconfiguration each. However since the three clients are separated by significant latencies, the reconfigurations are not as closely synchronized as in the LAN experiments, where all reconfiguring clients where located on the same machine.

Figure 6a shows normal and maximum read latency for this experiment. Figure 6b shows the overhead caused by 1 or 3 concurrent reconfigurations. Since the reconfigurations are not closely synchronized, the batching mechanisms fails to combine them. Thus, SM-Store, L-Rambo perform similar to Rambo. SpSn-Store performs worth than DynaStore for both a single, and concurrent reconfigurations. This is caused by the batching mechanism in SpSn-Store, which is performed by all clients, but has little effect in the WAN setting.

7 Conclusion

We have evaluated different algorithms for reconfiguration of atomic storage, both with and without consensus. For the different algorithms, we measure both reconfiguration latencies and the overhead caused by reconfigurations. Our experiments show that novel algorithms for reconfiguration without consensus perform similar to consensus-based L-Rambo, if the latter has a stable leader. However, especially SM-Store performs significantly better than Rambo, when the latter is run without a stable leader. Our experiments suggest that if *read* and *write* operations do not help concurrent reconfigurations to complete, that significantly reduces the overhead.

References

1 Marcos Kawazoe Aguilera, Idit Keidar, Dahlia Malkhi, and Alexander Shraer. Dynamic atomic storage without consensus. *J. ACM*, 58(2), 2011.

2 Masoud Saeida Ardekani and Douglas B. Terry. A self-configurable geo-replicated cloud storage system. In *OSDI 2014*.

3 Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, January 1995. doi:10.1145/200836.200869.

4 Keren Censor-Hillel, Erez Petrank, and Shahar Timnat. Help! In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, PODC'15, pages 241–250, New York, NY, USA, 2015. ACM.

5 Gregory Chockler, Seth Gilbert, Vincent Gramoli, Peter M. Musial, and Alex A. Shvartsman. Reconfigurable distributed storage for dynamic networks. *Journal of Parallel and Distributed Computing*, 69(1), 2009.

6 Gregory Chockler, Dahlia Malkhi, and Danny Dolev. A data-centric approach for scalable state machine replication. In André Schiper, AlexA. Shvartsman, Hakim Weatherspoon, and BenY. Zhao, editors, *Future Directions in Distributed Computing*, volume 2584 of *Lecture Notes in Computer Science*, pages 159–163. Springer Berlin Heidelberg, 2003.

7 Partha Dutta, Rachid Guerraoui, Ron R Levy, and Marko Vukolic. Fast access to distributed atomic memory. *SIAM Journal on Computing, Vol 39, N°8, December 2010*, 2010.

8 Jose M. Faleiro, Sriram Rajamani, Kaushik Rajan, G. Ramalingam, and Kapil Vaswani. Generalized lattice agreement. In *PODC 2012*, pages 125–134. ACM, 2012.

9 Rui Fan and Nancy Lynch. Efficient replication of large data objects. In Faith Ellen Fich, editor, *Distributed Computing: 17th International Conference, DISC 2003, Sorrento, Italy, October 1-3, 2003. Proceedings*, pages 75–91. Springer, 2003.

10 Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985. doi:10.1145/3149.214121.

11 Eli Gafni and Dahlia Malkhi. Elastic configuration maintenance via a parsimonious speculating snapshot solution. In Yoram Moses, editor, *Distributed Computing: 29th International Conference, DISC 2015. Proceedings*, pages 140–153. Springer, 2015.

12 S. Gilbert, N. Lynch, and A. Shvartsman. Rambo: a robust, reconfigurable atomic memory service for dynamic networks. *Distr. Comp.*, 23(4), 2010.

13 L. Jehl and H. Meling. Additional material. URL: `http://www.ux.uis.no/~ljehl/pdf/thecase.pdf`.

14 Leander Jehl, Roman Vitenberg, and Hein Meling. Smartmerge: A new approach to reconfiguration for atomic storage. In Yoram Moses, editor, *Distributed Computing: 29th International Conference, DISC 2015. Proceedings*, pages 154–169. Springer, 2015.

15 Leslie Lamport. On interprocess communication – part ii: Algorithms. *Distributed Computing*, 1(2):86–101, 1986.

16 Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, December 2001.

17 Jacob R. Lorch, Atul Adya, William J. Bolosky, Ronnie Chaiken, John R. Douceur, and Jon Howell. The smart way to migrate replicated stateful services. In *EuroSys*, 2006.

18 Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *SOCC*, 2012.

19 Cheng Shao, Jennifer L. Welch, Evelyn Pierce, and Hyunyoung Lee. Multi-writer consistency conditions for the shared memory objects. In *DISC 2003*.

20 Alexander Shraer, Jean-Philippe Martin, Dahlia Malkhi, and Idit Keidar. Data-centric reconfiguration with network-attached disks. In *LADIS 2010*.

21 Daniel Steinberg and Stuart Cheshire. *Zero Configuration Networking: The Definitive Guide*. O'Reilly Media, Inc., 2005.