Reconfiguring Replicated Atomic Storage: A Tutorial

Marcos K. Aguilera* Idit Keidar† Dahlia Malkhi*
Jean-Philippe Martin* Alexander Shraer†[1]

*Microsoft Research Silicon Valley
†Technion – Israel Institute of Technology

Abstract

We live in a world of Internet services such as email, social networks, web searching, and more, which must store increasingly larger volumes of data. These services must run on cheap infrastructure, hence they must use distributed storage systems; and they have to provide reliability of data for long periods as well as availability, hence they must support online reconfiguration to remove failed nodes and add healthy ones. The knowledge needed to implement online reconfiguration is subtle and simple techniques often fail to work well. This tutorial provides an introductory overview of this topic, including a description of the main technical challenges, as well as the various approaches that are used to address these challenges.

1 Introduction

Distributed storage systems have become increasingly important in data centers and other enterprise settings. Such systems comprise a network and several nodes (machines) that cooperatively store large amounts of information for a large number of users. By storing data redundantly, for example using *replication* over several nodes, the system can tolerate the failure of some of its nodes without losing data.

While it provides some level of reliability, replication by itself is not sufficient for safely guarding data over years or decades. This is because, as nodes fail, the system gradually loses its ability to tolerate further failures, until the point when it loses all replicas and data is lost. To address this problem, the system needs a way

[1]Author supported by an Eshkol Fellowship from the Israeli Ministry of Science.

Bulletin of the EATCS no 102, pp. 84–108, October 2010

to *reconfigure* its set of replicas, so that failed nodes can be replaced with healthy ones that restore the level of redundancy that once existed.

More precisely, node reconfiguration, or simply reconfiguration, is the act of changing the set of nodes that comprise the system. Reconfiguration is a key functionality for reliable distributed storage systems. Examples of its utility include the following:

- replacing failed nodes with healthy ones;
- upgrading software or hardware, by replacing out-of-date nodes with up-to-date ones;
- decreasing the number of replicas to free up storage space;
- increasing the number of replicas for recently written non-backed-up data;
- moving nodes within the network or the data center.

In this tutorial, we discuss the key issues in the reconfiguration of distributed storage systems. For concreteness, we focus the discussion on systems that support atomic read/write operations and employ majority replication [4] to tolerate crash failures. Majority replication is a common technique for implementing fault-tolerant atomic storage, while being conceptually very simple. By "fault-tolerant" we mean that the system remains operational most of the time despite failures; by "atomic" we mean that the system provides linearizability [14], a strong type of consistency that guarantees that a read returns the most recent version of data. With majority replication, each data item is assigned a number n of replicas, where a majority of them are kept up-to-date (they store the most recent version of the data), while a minority of replicas may fail or lag behind because they are slow or unresponsive. We explain the basics of majority replication in Section 2, which applies to systems that are *not* reconfigurable.

We start the discussion of reconfiguration in Section 3, by explaining what we mean by a system that supports reconfiguration. In particular, reconfiguration affects the *liveness* or availability guarantees of the system. With majority replication, the system remains live even if up to a minority of replicas crash. For example, with $n = 3$ replicas, we can tolerate one crash (a minority). If we reconfigured the system to have $n' = 7$ replicas, we could tolerate three crashes once reconfiguration is completed. But how many crashes can we tolerate if the crashes occur while the reconfiguration is in progress? And what happens if multiple reconfigurations were initiated and some have completed while others are still in progress?

The algorithms to provide reconfiguration are subtle and easy to get wrong, as we explain in Section 4. If not done correctly, reconfiguration can lead to a loss of

atomicity, whereby the storage system retrieves stale versions of data, rather than the most recently written version. To avoid such problems, during reconfiguration clients of the storage system must carefully coordinate to transition from the old to the new configuration as they store and retrieve data in the system. Moreover, when a new replica is added to the system, it must be carefully populated with fresh data so that it remains in sync with other replicas. This is tricky if reconfiguration can occur *online*, while the clients are actively trying to modify the data stored in the system.

The first reconfiguration algorithms to allow online operation required processes to reach consensus on the sequence of reconfiguration requests, to avoid the problems that may arise from concurrent handling of such requests. We explain how these algorithms work in Section 5. However, consensus is known to be impossible to implement in asynchronous systems subject to failures [10], and therefore these algorithms must rely on some amount of synchrony. In contrast, the replication algorithm itself is known to work in a completely asynchronous system. So, this naturally raises the question of whether reconfiguration really requires synchrony (or consensus) or not. This question was recently answered negatively, due to the existence of reconfiguration algorithms that work in asynchronous systems (without consensus), which we describe in Section 6.

While most of this tutorial focuses on tolerating crash failures, some systems may be subject to the more general *Byzantine* failures. We discuss the additional challenge of reconfiguration in this setting, and what to do about it, in Section 7. We then briefly cover other related work in Section 8, namely, state machines, dynamic quorums, and virtual synchrony. We conclude in Section 9.

2 Background: replicated atomic distributed storage

In this section, we explain the setting we consider and give some background on replicated atomic distributed storage systems.

2.1 Setting

We consider a distributed system with two types of nodes: *storage nodes* keep data for applications and users, while *client nodes* (or simply *clients*) issue requests to read and write data on storage nodes. The term *node* refers to a storage node or a client node.

A network allows nodes to communicate. We assume there is a link between every pair of nodes, which need not be a physical link.

Nodes may fail, and we are mostly concerned with *crash failures*, which causes a node to stop; we also briefly consider *Byzantine failures*, which causes a node to possibly deviate from its algorithm. If a node crashes, it stops executing its algorithm possibly without any advance warning to other nodes. Crashes are permanent: once a node stops, it never resumes execution. We make this assumption without loss of generality: if the hardware of the node recovers later, the hardware restarts as a new node in the system. If a node never fails it is called a *correct node*; otherwise, it is called a *faulty node*.

If the network fails by intermittently dropping or corrupting messages, these failures can be handled by standard mechanisms such as retransmissions and checksums. Thus, we can assume without loss of generality that messages are not dropped or corrupted. More precisely, we assume messages are unique (e.g., they have sequence numbers) and links are *quasi-reliable*, meaning they satisfy two properties: (1) if a correct node p sends a message m to a correct node q then eventually q receives m from p, and (2) a node q receives a message m from a node p at most once, and only if p previously sent m to q. We do not consider permanent network failures.

We do not assume any bounds on the relative speed of nodes or on message delivery delays. Technically, the system is *asynchronous* or *partially synchronous* [9].

2.2 Requirements of an atomic storage system

The storage system provides two operations, *write* and *read*, for storing and retrieving data, respectively. Typically, the storage space is divided into several storage cells, such as blocks, sectors, or files, to help users organize and manage their data. We are interested in the fundamental storage algorithms and techniques, and these are exactly the same for each storage cell. Therefore, we assume without loss of generality that the storage system comprises a single storage cell. We denote the operation to write a value v to this cell as *write*(v) and the operation to read this cell as *read*().

The storage system should provide the following properties:

- *Safety.* We consider a type of consistency called atomicity or linearizability [14], which states that each operation must appear to take place instantaneously between the time when it is invoked and the time when it produces a response.

- *Liveness.* A correct client should terminate its operation despite the failures of a reasonable number of client nodes and/or storage nodes.

Note that liveness is ensured only if the failures are limited in number, where the exact number is a feature of the protocol and it is called the protocol's *resilience*. The majority replication we present below allows the failure of any number of client nodes and of any minority of the storage nodes. When the set of storage nodes can be reconfigured, the number of allowed failures depends on the occurrence of reconfiguration operations, as we explain in Section 3. For instance, the resilience generally increases if we add storage nodes.

2.3 Majority replication

Majority replication is a simple scheme that provides the properties above by storing and retrieving data at a majority of storage nodes, while a minority of such nodes may crash.[2] We explain majority replication [4] in a simple context *without* reconfiguration, where the set of storage nodes is *static* and known a priori when the system is started. The rest of this tutorial will explain how we can reconfigure the set of storage nodes without affecting the basic workings of the protocol we describe in this section.

The key intuition behind majority replication is that any two majorities of storage nodes always have at least one storage node in common. Therefore if some client stores value v at a majority of storage nodes then another client is guaranteed to see v when it queries any majority. We now explain the scheme in more detail.

Each storage node keeps a local copy of what it believes to be the most recent value stored by a client, together with a timestamp indicating the freshness of the value. A *vt-pair* refers to a pair of (*value*, *timestamp*), which a storage node keeps. To execute a *write*(v) operation, the client proceeds in two phases:

- *Get phase.* The client asks storage nodes to send their vt-pairs and waits for a majority of responses. The client finds the largest received timestamp, and then chooses a higher unique timestamp *ts-new*. Uniqueness can be ensured by adjoining the client-id to the timestamp, so that a timestamp consists of a pair with a number and a client-id, ordered lexicographically.

- *Set phase.* The client asks storage nodes to store the vt-pair (v, *ts-new*). Each storage node checks if this vt-pair has a larger timestamp than the one it stores; if this is so, the storage node stores the new vt-pair. In either case, the storage nodes sends an acknowledgement to the client. The client then waits for a majority of acknowledgements. The *write*(v) operation then returns an *ok* response.

[2] Majority replication can be generalized to quorum-based replication as we explain in Section 8.

To execute the *read*() operation, the client also executes two phases:

- *Get phase.* The client asks storage nodes to send their vt-pairs and waits for a majority of responses. The client finds the largest received timestamp and selects the vt-pair (v, t) with that timestamp.
- *Set phase.* The client asks storage nodes to store the vt-pair (v, t). Each storage node checks if this vt-pair has a larger timestamp than the one it stores; if this is so, the storage node stores the new vt-pair. In either case, the storage nodes sends an acknowledgement to the client. The client then waits for a majority of acknowledgements. The *read*() operation then returns v as its response.

The Set phase in *read*() is needed to prevent oscillating reads, in which successive reads oscillate between an old and a new value while a write is in progress—which is a violation of atomicity. The Set phase ensures that a subsequent *read*() will return a value at least as recent as the value returned by the current *read*() [4].

We observe that the protocols for *read*() and *write*(v) are very similar; the key difference is the vt-pair used in the Set phase: for *write*(v), it consists of v and a new timestamp, while for *read*(), it consists of the vt-pair from the Get phase with highest timestamp.

It is easy to see that this protocol can tolerate the crash of a minority of storage nodes and an arbitrary number of client nodes, and it works in an asynchronous system.

3 Reconfiguration service and liveness

A configuration is the set of storage nodes that comprise the storage system. In a static system, the configuration does not change and it can be hard-coded into the protocols that clients use. In a reconfigurable system, there are ways to change the configuration via a *reconfiguration operation*, which we explain in Section 3.1. This operation is invoked by clients, which can execute on behalf of an administrator, a monitoring system, or any other entity that needs to reconfigure the system. In Section 3.2, we describe a service that allows new clients to discover the current configuration, so they can start using the storage service. In Section 3.3, we give a specification of the liveness conditions of a reconfigurable system as a function of how it has been reconfigured.

3.1 Reconfiguration interface

To allow reconfiguration, we augment the storage system interface with an operation *reconfigure*(*addSet*, *removeSet*) that clients can invoke to add and/or remove

storage nodes. We allow *removeSet* to include nodes that are not currently in the system, or *addSet* to include nodes that are already in the system—in both cases, these nodes are simply ignored. The *reconfigure* operation returns an *ok* response.[3] The implementation of *reconfigure* may continue executing actions in the background after a response has been returned.

It is desirable to allow clients to invoke *reconfigure* at any time and, in particular, concurrently with the execution of read/write operations or even other *reconfigure* operations. The reason is that, in a system that is not synchronous, it is generally impossible to ensure that an operation is running by itself. For example, if one client invokes *reconfigure* and then crashes, it is impossible for others clients to know that the crash has occurred: for all they know, the client may be merely slow and the *reconfigure* operation may still be running. If we did not allow the execution of concurrent *reconfigure* operations, this scenario would forever preclude any subsequent reconfigurations. (An alternative is to employ a human operator to ensure operations execute by themselves, but this solution could be expensive and error prone.)

The reconfiguration operation does not affect the semantics of read and write operations. However, as we explain in Section 3.3, it generally affects the liveness of the system or, more specifically, number of failures that the system can tolerate before it becomes unavailable.

3.2 Directory service

When a new client starts executing, it needs a way to bootstrap itself and learn the identity of storage nodes so that it can access the storage system. This information is provided by a *directory service*, which upon request returns a *hint* of what are the storage nodes in the current configuration. The directory service can be updated by the reconfiguration scheme once a new configuration emerges. We allow the directory service to return storage nodes in old configurations, say because it is lagging behind the storage service; in that case, the clients will be able to contact nodes in the old configurations and learn the latest configuration via the protocols that we describe later. However, if the directory service returns storage nodes from a very old configuration, these storage nodes may be unresponsive because they have crashed or were taken down, and this information is of no use to the clients. Thus, we make the assumption that eventually the directory service catches up and returns storage nodes in a recent configuration. More precisely, if there are finitely many executions of reconfiguration operations, then there is

[3]There is an alternative interface for *reconfigure*, in which the operation specifies the entire set of storage nodes in the new configuration, and the operation *could* (but it is not guaranteed to) return a *fail* response when multiple reconfigurations occur simultaneously. We believe the interface we give is better suited for systems with concurrent operations.

time after which the directory service returns the storage nodes in the *final configuration*, which is the configuration that incorporates all the changes specified by the reconfiguration operations. The directory service can be implemented via state machine replication or, in practice, it can be provided by the DNS service by choosing some fixed DNS names to map to the IP addresses of the storage nodes.

3.3 Specifying liveness

A storage system has a maximum number of failures that it can tolerate (its resilience). Beyond this number, data can be lost and the system will cease to be live, that is, clients operations will fail to complete.[4] We are interested in specifying the resilience conditions under which the system is guaranteed to be live. In a static system, these conditions are often simple; for example, with majority replication, the system can tolerate the crash of any minority of storage nodes—that is, the system remains live as long as a majority of storage nodes are correct.

In a reconfigurable system, things are more complicated, because reconfiguration modifies the set of storage nodes. An elegant way to specify the liveness condition is in terms of the reconfiguration operations themselves [2], by taking into consideration the operations that have completed (i.e., those that have returned an *ok* indication) and operations that are still outstanding (i.e., those that have started but have not yet returned *ok*). Another approach to specify liveness is in terms of the internal actions of the reconfiguration protocol. This other approach is less elegant since it does not separate specification and implementation, so we do not consider it here.

When there are no outstanding reconfiguration operations, the liveness condition should match that of a static system: liveness should require only a majority of storage nodes in the current configuration to be correct. But what happens with liveness while reconfiguration operations execute? In general, removing a live storage node will worsen resilience, while adding a storage node will improve resilience. The key intuition is that, while the reconfiguration operation is executing, its changes may or may not have taken effect and, in the worst case, we must assume that the changes that worsen resilience have occurred, while the changes that improve resilience have not yet occurred.

We must assume that clients cannot continually issue reconfiguration operations, because otherwise these reconfigurations may forever hinder the execution of read or write operations. For example, suppose that a client starts a *write*(v) operation. Before the write completes, it must store v at a majority storage nodes. But if other clients continually issue reconfiguration operations to add new storage

[4] Some storage systems can be live even when data is lost, by returning stale data or other garbage to the clients. We do not consider such systems in this article.

nodes, it is possible that v never gets stored in a majority of this growing system, and so the write operation may never complete. Therefore, we must assume that there is only a finite number of executions of the reconfiguration operation.

We now explain more precisely the liveness condition. We give a weak property in the sense that it should be satisfied by any reasonable reconfiguration protocol, but certain protocols may ensure a stronger liveness condition. We define four time varying sets: at a given time t in the execution, $Current(t)$ is the configuration that reflects all reconfigurations that completed by time t. Nodes added (resp., removed) in reconfigurations that were invoked but did not complete by time t belong to $AddPending(t)$ (resp., $RemovePending(t)$). Finally, nodes that crash by time t are in $Failed(t)$. Note that these sets are not known to the nodes; they are defined from the point of view of an omniscient global observer.

Before we use these sets to specify the liveness of a reconfigurable system, let us restate the liveness property of static systems in terms of these sets. Here, $AddPending(t)$ and $RemovePending(t)$ are always empty. The liveness requirement can be stated as follows:

> **Static liveness.** If at any time t in the execution, fewer than $|Current(t)|/2$ processes out of $Current(t)$ are in $Failed(t)$, then all the operations of correct clients eventually complete.

We expect a *reconfigurable* system to satisfy this property whenever there are no outstanding reconfigurations, i.e., when $AddPending(t)$ and $RemovePending(t)$ are empty. Intuitively, this means that only failures in the current configuration are relevant and the overall number of failures in the execution is immaterial.

What about the case when there are outstanding reconfigurations? Note that, in a distributed system, different nodes might have different perceptions of the current configuration. For example, suppose there is an outstanding reconfiguration operation to add s_a and remove s_r. There may be a time when some node already considers s_a to be in the new configuration, while another does not include it yet. At that time, to be safe, we need to take both possibilities into account. To accommodate the view that s_a is in the configuration, we should count a failure of s_a as part of our failure budget (which is a minority of the current configuration). On the other hand, we need to accommodate the view that s_a is not yet in the new configuration. Therefore, we do not allow more failures than a minority of $Current(t)$, which does not include s_a. Similarly, the node s_r being removed should already be counted towards the budget of failed nodes, because some nodes may no longer consider it to be in the current configuration. We arrive at the following definition [2]:

> **Dynamic liveness.** Suppose there is a finite number of reconfigurations. If at any time t in the execution, fewer than $|Current(t)|/2$ processes out of

Current$(t) \cup$ *AddPending*(t) are in *Failed*$(t) \cup$ *RemovePending*(t), then all the operations of correct clients eventually complete.

It is easy to see that this liveness condition requires that at any given time, the set of crashed processes and those whose removal is pending comprises no more than a minority of the current configuration, or of any pending future configuration.

As an aside, we note that the liveness specification is slightly more complex in systems where a client node must be one of the storage nodes [2], because in this case we need to worry about whether a client is in the system (i.e., the storage node is in the configuration). We do not consider this setting here.

4 The dangers of naïve reconfiguration

In a nutshell, reconfiguration entails three steps: (1) *populate* at least a majority of storage nodes in the new configuration with the current version of the data, (2) *stop* the storage nodes that are to be removed, (3) *inform* clients about the new configuration so that they subsequently access the new set of storage nodes.

The simplest reconfiguration scheme is to take down the storage system so that there are no client operations in execution, and then perform each step above sequentially. In the populate step, an administrative node executes the Get phase (Section 2) using the nodes in the old configuration to learn the current version of data, followed by the Set phase using the nodes in the new configuration to store the data; if the administrative node crashes while executing these actions, another administrative node restarts from the beginning. This approach is simple, but it leads to unavailability of the storage system during reconfiguration, which could be unacceptable. (We observe that storage systems in practice tend to have many storage cells and lots of data to be populated, so reconfiguration can take a long time.) A similar approach would be to place a write lock on the old configuration, to prevent clients from writing to it, while the three steps above execute. This approach would have similar unavailability problems: clients are not able to write for long periods. We are interested in *online* reconfiguration schemes, which perform reconfiguration while the storage system remains accessible to clients.

However, inconsistencies may arise if we naïvely execute the three reconfiguration steps described above (populate, stop, inform) while clients continue to execute their operations. We give three scenarios that illustrate how things can go wrong. The first two scenarios arise when *read* and *write* operations occur while the system is reconfiguring. The third deals with multiple *reconfigure* operations that execute concurrently.

In the first two scenarios, the system initially has three storage nodes s_1, s_2, s_3 (the old configuration) and there is a reconfiguration operation to replace s_2 with s'_2

(i.e., to remove s_2 and add s'_2), so the system ends up with storage nodes s_1, s'_2, s_3 (the new configuration). Because the system is not synchronous, it is impossible to ensure that clients are informed simultaneously about the new configuration. Thus, at a given time, different clients may be executing with different configurations. The following problems can occur:

1. Suppose one client writes new data using the new configuration, and another reads the old one. The first client stores a new value in two (a majority of) storage nodes, say s_1, s'_2. Subsequently, the second client, which reads using the old configuration, may query two storage nodes, say s_2, s_3. The second client misses the new value, causing the client to read stale data—a violation of atomicity.

2. Similarly, consider the client that is performing the reconfiguration, and suppose it is populating the new configuration with value v, which it obtained from s_1, s_2 in the old configuration. Suppose the client stores v in s'_2, s_3, a majority of the new configuration. Then, another client starts to write v' and it is still using the old configuration, so it stores v' in s_2, s_3, a majority of the old configuration. Now all clients switch to the new configuration, but because s'_2 is populated with v and s_1 still has v, a subsequent read may query storage nodes s_1, s'_2 and miss the new value v'—a violation of atomicity.

In the third scenario, the system initially has four storage nodes $s_1, \ldots, s_4$ and two clients wish to reconfigure concurrently. The following problem can occur:

3. One client starts a reconfiguration operation to remove s_4, while a second client starts a reconfiguration operation to add s_5. The first client succeeds and its new configuration is s_1, s_2, s_3, while the second client also succeeds and its new configuration is s_1, s_2, s_3, s_4, s_5. The clients will learn of each other's reconfiguration later, but they have not yet done so. Now, the first client writes a value v' to s_1, s_2, which is a majority of its new configuration; subsequently, if the second client reads using s_3, s_4, s_5 then it misses v'—a violation of atomicity.

This last example is an example of a "split brain" scenario. Such conflicting reconfigurations could arise due to asymmetric network delays. Early attempts to solve the reconfiguration problem were susceptible to this problem, as described in [23].

In the next sections, we present solutions that avoid all three problems, by coordinating reconfigurations and making the *read* and *write* operations aware of reconfigurations.

5 Reconfiguration using consensus

To avoid the "split brain" problem of Section 4, one solution is to order the reconfiguration operations in a way that all nodes agree. This idea dates back to work on dynamic replica replacement in distributed database systems [13, 15], which used two-phase commit for reconfiguration. More recent solutions [18, 12, 7, 22] use *consensus* [17], which is a building block that allows nodes to agree on a value. More precisely, with consensus, each correct node proposes some value and must reach an irrevocable decision on a value, such that nodes never decide on different values, and the decision value is one of the proposed values.

Consensus is used to establish a sequence of configurations that all nodes agree upon. More precisely, there is a consensus instance associated with each configuration c, allowing the nodes in c to agree on the subsequent configuration. To execute a reconfiguration operation, a client tries to cause the consensus instance of the latest configuration c to decide on its new configuration (by requesting the nodes in c to propose the new configuration). If the client is not successful, there must have been another reconfiguration operation that succeeded, creating another configuration; the client retries using the other configuration, until it is successful. In this way, consensus ensures that each configuration is followed by a single configuration. For example, in the third scenario of Section 4, the current configuration is $s_1, \ldots, s_4$ and there are two outstanding reconfiguration operations; consensus ensures that only one of them is chosen to be the next configuration after $s_1, \ldots, s_4$.

Consensus solves the "split brain" problem, but we must still avoid the other two bad scenarios described in Section 4. They arise because some clients use the old configuration to read or write, while other clients use a new configuration. The problem here is not that clients disagree on the ordering of configurations, but that some clients may not yet be aware of the last configuration in the ordering (e.g., because when they checked, the latest configuration had not been established yet). This problem is solved as follows. As a first step, the reconfiguration operation stores a forward pointer at a majority of nodes in the old configuration, which points to the new configuration. This pointer ensures that, if a client were to perform an operation on the old configuration, it will find the forward pointer and will execute its operation on the new configuration as well. As a second step, the reconfiguration operation populates the new configuration, so that it stores data as recent as the old configuration. As the third and final step, clients can be told to execute their reads and writes only on the new configuration. Even if some clients were told before others, this will not violate atomicity: clients that were told will use the new configuration, while the others will use both the old and the new configuration, due to the forward pointer.

For example, consider the first two scenarios of Section 4. The reconfiguration

operation causes a majority of nodes among s_1, s_2, s_3 to store a forward pointer to s_1, s_2', s_3. With this pointer, clients will execute reads and writes using both the old and new configurations (they may start with the old configuration only, but will see the forward pointer). The reconfiguration operation then populates the new configuration; then, it tells clients to disregard the old configuration and use only the new configuration.

Different algorithms implement these ideas in different ways. We now explain one such algorithm, whose main ideas come from the RAMBO protocol [18, 12, 11]. Each storage node q stores a vt-pair (see Section 2.3) and a local variable $cmap_q$, which maps indices to configurations. The map always consists of a prefix of *retired* configurations, followed by one or more active configurations, followed by an infinite suffix of yet-unknown configurations. To store the map compactly as a finite data structure, the initial prefix of retired configurations and the final suffix of unknown configurations are not represented explicitly. Each client also stores a *cmap*; below we explain how the client obtains the map initially. To execute a read or write, the client contacts storage nodes in the active configurations according to its *cmap*. Storage nodes piggyback their *cmaps* to the messages to the client, which integrate with its own *cmap*. The protocol for reading and writing a value works roughly as follows:

- *Get phase.* When a client p wants to read or write the object, it sends a request to the storage nodes in the active configurations in its $cmap_p$. Each storage q replies with its vt-pair (v_q, t_q) and its $cmap_q$. The client integrates $cmap_q$ into its own $cmap_p$, including retiring configurations that are retired in $cmap_q$. It repeats its request to any new storage nodes in active configurations that it learns about. This phase completes when the client has obtained replies from a majority of storage nodes in every active configuration in $cmap_p$ (this is called a *fixed point*). The client then finds the vt-pair (v, t) with largest timestamp t.

- *Set phase.* The client first picks a vt-pair as follows. If the operation is to read, the picked vt-pair is (v, t) (the vt-pair found in the Get phase). If the operation is to write a value v', the picked vt-pair consists of v' and a unique timestamp t_p higher than t.

  The client sends the picked vt-pair to the storage nodes in the active configurations in $cmap_p$. Upon receiving the vt-pair (w, t'), the storage node updates its vt-pair (v_q, t_q) if $t' > t_q$, and then sends back an acknowledgement to the client. The acknowledgement piggybacks the storage node's $cmap_q$. The client integrates $cmap_q$ into its own $cmap_p$, and sends the picked vt-pair to any new storage nodes it learns about. As in the Get phase, this phase

terminates at a fixed point, when the client has received acknowledgements from a majority of storage nodes in each active configuration in $cmap_p$.

A new client that just started has an empty *cmap*; similarly, a client that has been inactive for a while may have a very old *cmap* that contains only configurations with crashed storage nodes. This is problematic, because these clients are unable to communicate with any storage nodes. To address this problem, we rely on the directory service (Section 3.2). Each client periodically contacts the directory service and obtains a list of storage nodes. If this list has nodes that are not in active configurations in the client's *cmap*, or if the client just started executing, the client requests the *cmap* from each of those nodes. Upon receiving a *cmap*, the client integrates it with its own. Note that the directory service may return nodes from old configurations, but as we explained in Section 3.2, eventually it must return nodes from the final configuration.

Reconfiguration and garbage collection. Reconfiguration relies on a separate module to obtain consensus on a sequence of configurations. This module could be implemented using consensus algorithms, or it can be provided manually by human that decides on a sequence of configurations. When the module appends a new configuration to the sequence, the directory service (Section 3.2) is informed of the new configuration.

For a reconfiguration operation to complete, the old configuration must eventually be retired or garbage collected, meaning that clients can disregard the old configuration and execute operations using only the newer configurations (and therefore servers in the old configuration that are not in the newer configurations can be turned off). If c is a configuration and c' is the next configuration in the sequence, then c can be garbage collected after two actions have been performed:

- *Action 1.* The next configuration c' has been stored in the local *cmap* variables of a majority of storage nodes of c. Intuitively, this provides the forward pointer from c to c', so that any operations that execute using configuration c will discover the existence of c' (and therefore will execute using c' as well).

- *Action 2 (State transfer).* The Get and Set phases were performed after Action 1. This action guarantees that any value previously stored only in c will be copied to c'.

It is worth making a note about liveness. Before the garbage collection is completed, a majority of nodes in c and a majority of nodes in c' must be alive. After c is garbage collected, we only need a majority of c' to be alive. At that

point, even if all the nodes in c crash, this is not a problem since the knowledge held by c was transferred to c'.

The RDS algorithm [7], developed after RAMBO, is based on the same principles, but it integrates the consensus algorithm into the garbage collection protocol to improve efficiency, at the cost of having no flexibility of employing other consensus implementations.

6 Reconfiguration without consensus

Consensus cannot be implemented in asynchronous systems subject to node failures [10], and therefore reconfiguration protocols that use consensus require some synchrony to work. In contrast, as we saw in Section 2.3, in a static system distributed storage can be implemented without any synchrony. Thus, the question arises whether reconfigurable storage really requires synchrony or not. Recently, it has been shown that it does not [2]—in particular, DynaStore is a scheme that works in asynchronous systems, without the need for consensus to prevent split-brain scenarios and other problems. We now briefly describe the main ideas in DynaStore. Our description is based on a variation of DynaStore presented in [22], as the original DynaStore protocol was designed for a model where no distinction is made between clients and storage nodes.

If different reconfiguration proposals are made concurrently, DynaStore does not attempt to agree on a single next configuration to follow each configuration. Instead, for each configuration c, each client has a *family* of next configurations that it believes could follow c, and the coordination mechanism guarantees that such families at different clients always intersect. Thus, there is at least one *common configuration* that all clients consider as a possible next configuration after c. Intuitively, clients will execute read and write operations on all configurations that they believe could follow c, to ensure that all clients overlap in the common configuration.

To ensure the existence of this common configuration, DynaStore uses the abstraction of a "weak snapshot". Each configuration c is associated with a weak snapshot, which is implemented using the storage nodes in c; the weak snapshot holds information about the family of possible configurations that follow c. A weak snapshot abstraction supports two operations: *update*(d) and *scan*(). *update*(d) is called to propose a configuration change d to configuration c, where d consists of sets of nodes to be added and removed; and *scan*() is called to retrieve a subset of the changes to configuration c previously proposed by storage nodes. Weak snapshots ensure that (1) once an update completes, every subsequent scan returns a non-empty set of configuration changes; and (2) once some configuration change is returned by a scan, it is returned by all subsequent scans.

Moreover, there exists some common configuration change d_{common} that is returned in all non-empty scans. This is called the *non-empty intersection* property. Intuitively, d_{common} can be thought of as the first configuration change applied to c. The implementation of weak snapshots ensures liveness provided that at most a minority of the storage nodes in c are removed or fail.

The weak snapshot of a configuration c is implemented as follows. Each storage node s_i in c stores an array N_i indexed by the set of storage nodes in c. The set of such arrays forms a matrix N distributed over the storage nodes. $N_i[j]$ holds the configuration change endorsed by storage node s_j, if it is known to s_i. We now explain how the client executes the two operations, update and scan, of the weak snapshot of configuration c:

- To execute *update*(d), a client first tries to get one of the storage nodes in c to endorse the configuration change d. To this end, it sends an endorsement request with d to the storage nodes in c and waits for a response. A storage node s_i endorses the first configuration change d it receives for c, by storing d in $N_i[i]$ and responding with d. If a storage node had already endorsed another configuration change d', it simply responds with d'. When the client receives the first response d'' from some node s_j, it ensures that a majority of the storage nodes know that s_j endorses d'',[5] by executing essentially the Set phase from Section 2.3 on the j-th column of the matrix N, that is, on $N_*[j]$.

- To execute *scan*(), a client collects the arrays from a majority of the nodes (this is done similarly to a *read()* in Section 2.3), and then does it again to guarantee the non-empty intersection property; it returns a set containing all configuration changes it obtained.

The algorithm of Section 5 uses consensus to establish a sequence of configurations that all nodes agree upon. In contrast, DynaStore uses weak snapshots and, instead of establishing a sequence, it establishes a DAG (directed acyclic graph) of configurations. Edges in the DAG correspond to configuration changes—a client creates an edge from a configuration c by executing the update operation of the weak snapshot of c; and it checks what edges are outgoing from c by executing the scan operation on the weak snapshot of c. Scans performed by different clients may return a different set of edges, and so different clients may obtain different subgraphs of the DAG. However, the non-empty intersection property ensures that there is a path that is common to all the subgraphs obtained by clients. Clients do not know what is this common path.

[5] If $d'' \neq d$, then d may not have been stored in the weak snapshot of c; intuitively, this is fine since there is no point in changing an old configuration. As we explain later, the client will then try to execute *update*(d) on a newer configuration.

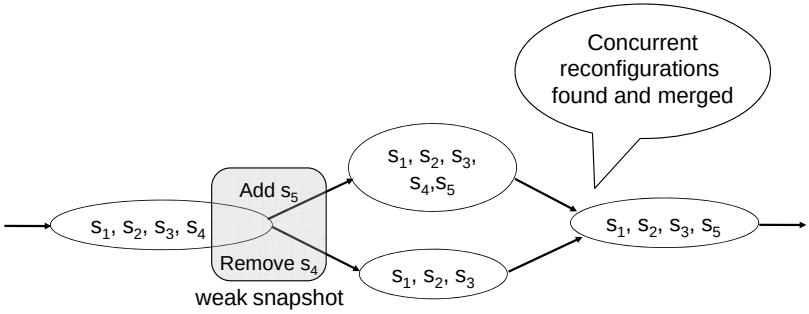

Figure 1: Example DAG of configurations for third scenario of Section 4.

During a *read*, *write*, and *reconfigure* operation, the client traverses the DAG of configurations. If the client finds a vertex with multiple outgoing edges (several branches), the client reconciles these branches. To do so, the client creates a new vertex that merges the branches (i.e., the new vertex includes all of the configuration changes in the branches), and creates edges from each branch to the new vertex, by executing the update operation on the weak snapshots of the branches. The client may fail to add some edges—recall that the update operation is not required to create a new edge if another edge already exists in the weak snapshot; in that case, the client follows this other edge to reach a new vertex, and then tries to add an edge from that vertex. Essentially, DynaStore guarantees atomicity by reading from the configurations in all possible paths in the DAG, using the Get phase of Section 2.3 to read at each configuration. When a vertex without outgoing edges is reached, data is stored at this configuration using the Set phase of Section 2.3; then a scan is performed on the weak snapshot of this configuration to check for new edges that may have been created by concurrent reconfigurations. If there are new edges, the traversal continues. Otherwise, the configuration is guaranteed to appear on all paths seen by other operations, and the client can complete its operation—this will happen provided that (a) there is a finite number of reconfigurations, and (b) the other liveness conditions specified in Section 3 are met so that a majority of storage nodes are responsive in each traversed configuration. Note that the client ensures that all configuration changes encountered during the DAG traversal are reflected in the last traversed configuration (this is ensured by creating new vertices and edges as we described above).

Figure 1 illustrates a DAG that may be created by an execution of DynaStore for the third scenario of Section 4. Two conflicting reconfigurations execute concurrently and update the weak snapshot of the initial configuration s_1, s_2, s_3, s_4. After the updates, both reconfigurations scan the weak snapshot to find the outgoing edges. The non-empty intersection property ensures that both operations observe at least one common configuration change. For instance, it is possible

that the reconfiguration removing s_4 does not see the one adding s_5 and completes after transferring the state to configuration s_1, s_2, s_3; however, in this case, if the reconfiguration adding s_5 completes, it will notice and merge the branch in the DAG by creating a new vertex for s_1, s_2, s_3, s_5, connecting it to the DAG by updating the weak snapshots of the preceding branched configurations, and transferring the state to s_1, s_2, s_3, s_5.

As described above, during the traversal of the DAG of configurations, the client performs Get and Set to read and write the data in each configuration, and it performs *update* and *scan* to access the weak snapshot of that configuration. DynaStore guarantees correct interleaving of these data and snapshot operations: essentially, for every configuration in the DAG, if a Get is performed in the configuration concurrently with a Set, then either the client performing Get sees the data of the client performing Set, or the client performing Set sees at least all those outgoing edges in the DAG seen by the client performing Get; in the latter case, and if such outgoing edges exist (representing configuration changes proposed for the current configuration), the client performing Set continues traversing the DAG so that it performs Set in the latest configuration of the system.

A client starts executing an operation by communicating with the storage nodes in the most recent configuration it knows. It then traverses the DAG of configurations in the way we described above. It is possible that a configuration discovered during the traversal is old and no longer has a majority of responsive nodes. To handle such situations, the client periodically queries the directory service, obtaining a set of storage nodes; it asks these storage nodes for their latest configuration; if this configuration is new to the client, it retries its operation using the new configuration.

A recent study [22] compared a consensus-based reconfiguration algorithm against an asynchronous reconfiguration algorithm. This study discovered that each algorithm works best in a different case. When multiple reconfigurations are expected to occur concurrently and the latency of reconfigurations is important, the asynchronous algorithm is better, since consensus may take a long time to complete—theoretically, it may never complete [10]. On the other hand, with the consensus-based algorithm, no reader or writer is affected by concurrently ongoing reconfigurations until a new configuration is agreed upon, and only after this point in time do operations need to start working in the new configuration. With the asynchronous algorithm, operations must take into account reconfiguration proposals made concurrently (this means traversing the DAG of configurations for each operation). This results in slower reads and writes when multiple reconfigurations are operating concurrently.

7 Dealing with Byzantine failures

With Byzantine failures, we must assume a higher fraction of correct nodes than with crash failures (e.g., in a static setting, there must be more than two-thirds of correct nodes instead of a majority). Furthermore, if this assumption is violated, the system may lose both safety and liveness, whereas with crash failures, it may only lose liveness.

To deal with Byzantine failures, the storage system must rely on some techniques that are unrelated to the reconfiguration algorithm. First, the replication scheme must store data on more than two-thirds of the storage nodes to tolerate lies (e.g., see [19]). We call these sets with more than two-thirds of nodes *supermajorities*. Second, the consensus protocol, if used, must be resilient to Byzantine failures.

A challenge unique to reconfiguration with Byzantine failures is that storage nodes can lie about the composition of the current configuration, resulting in the "I still work here" attack, which we now describe.

7.1 The "I still work here" attack

A system designer typically makes assumptions on the number of machines that may be Byzantine (e.g., at most a third of them), based perhaps on the fact that someone maintains those machines. One of the purposes of reconfiguration is to relinquish machines and not have to worry about them anymore. After a storage node is removed from the configuration, there should be no assumptions on what can happen to it—in particular, it could crash or be subject of other types of Byzantine behavior.

A client may be inactive for a long time, and when it returns to activity, the set of storage nodes may have changed significantly, and the client may interact with storage nodes that were long removed from the system. These storage nodes could be Byzantine and pretend that the reconfiguration never took place, and therefore hijack the client requests and serve them inappropriately (e.g., by returning bogus results to reads).

One can avoid this attack by requiring that any new configuration overlaps with the old one in some number of machines (as in the work on dynamic Byzantine quorums [3]), but this goes against our goal of allowing arbitrary reconfiguration. Another solution is to assume one never-changing trusted server (such as a DNS server) that always points to the current configuration, but this violates our desire to allow removal of any part of the system.

7.2 The Forgetting Protocol

The above attack can be thwarted using cryptography [20]. The main idea is that, as mentioned above, most Byzantine fault-tolerant protocols assume that some fraction of the storage nodes are correct, at least while they are members of the system. A command that will remove nodes from the system can also ask them to overwrite the value of some variables that are needed to participate in the protocol. Correct nodes, by definition, will comply. Even if those nodes later turn Byzantine, it will be too late for them to recover those variables, since they were properly erased when the node was correct. Those variables contain cryptographic secrets in the form of keys that are sent anew every time some nodes are added to or removed from the system. One challenge is that, in an asynchronous system, the messages containing those keys may be delayed for a long time and arrive to the destination after it has become Byzantine: so the protocol must be designed to make sure that enough key material was deleted that removed nodes cannot take advantage of those delayed messages.

Allowing any participant to reconfigure the system would allow Byzantine participants to move the system to adversary-controlled machines. Instead, the protocol assumes that a trusted administrative entity selects a sequence of configurations. It can be a trusted person or a replicated state machine, akin to the consensus-based approach described in Section 5. The protocol then works as follows. Each storage node i holds a symmetric key s_i^t for configuration number t. It is computed by hashing the key from the previous computation. That is, $s_i^t = h(s_i^{t-1})$. The key is known only to that storage node and to the administrator, and the storage node keeps only the key corresponding to the current configuration. Each storage node keeps also an asymmetric key pair $(pub_i^t, priv_i^t)$ associated with the configuration, and a matching certificate from the administrator $cert = \langle i, meta, pub_i^t \rangle_{admin}$ (angle brackets indicate a signed message). The meta-data *meta* includes the configuration number, t.

When the administrator reconfigures the system to a new configuration t, it sends the following message to each storage node i: $(NEW\_VIEW, t, oldMeta, encrypt((cert, priv_i^t), s_i^t))$. Here, $encrypt(m, k)$ encrypts m using the symmetric key k. When a storage node sees that it is not part of the new configuration, it discards its certificate, private key, and symmetric key.

As part of the write and read protocols, clients receive a copy of the administrator's certificate, and a proof that the storage node has the corresponding private key. Clients only proceed when a supermajority of storage nodes have the key for the same configuration. In the protocol, every pair of supermajorities of nodes intersects at a correct node, so after all nodes in a supermajority are asked to delete their key, in every supermajority at least one storage node is missing its key. The protocol requires a supermajority to present their keys before proceeding with a

read or write, so no client will be fooled into using an old configuration.

The notion of protecting against removed nodes turning Byzantine has similarities with the notion of protecting against nodes slowly turning Byzantine over time. Proactive recovery [24, 6] addresses the latter problem: periodically rebooting machines to return them to a safe state, and using cryptographic techniques such as threshold cryptography to protect against an adversary squirreling away key material as it compromises machines. Threshold cryptography can be used to build a replicated state machine that can sign its consensus decisions (e.g. a reconfiguration command) without allowing any single node to learn the private key.

8 Related work

In this section, we briefly explain some work related to reconfiguration of atomic storage that we did not cover in the previous sections.

Applications. While reconfiguration of atomic storage started as a theoretical endeavor, this work has recently seen application to real systems. In particular, FAB [21] is a replicated storage system built using commodity machines that are prone to failures. FAB replicates data using a modification of the algorithm in Section 2 to ensure a property stronger than linearizability [1], and its reconfiguration uses the Rambo algorithm described in Section 5.

Dynamic quorum systems. A quorum system consists of a collection of sets such that any two sets in the collection have non-empty intersection. This concept can be used to replace majorities in majority-based replication, to obtain schemes that can tolerate the failure of more than a majority of some chosen set of nodes, at the expense not being able to tolerate the failure of any minority. For example, consider the following quorum system for five storage nodes $s_1, \ldots, s_5$:

$$
\begin{array}{llllll}
\{\{ & & & s_4, & s_5 & \}, \\
\{s_1, & s_2, & & & s_5 & \}, \\
\{s_1, & & s_3, & & s_5 & \}, \\
\{ & s_2, & s_3, & & s_5 & \}, \\
\{s_1, & s_2, & & s_4 & & \}, \\
\{s_1, & & s_3, & s_4 & & \}, \\
\{ & s_2, & s_3, & s_4 & & \}\}
\end{array}
$$

We can modify the majority-based scheme in [4] so that, instead of waiting for a majority of responses, processes wait to receive responses from one of the

sets above, say $\{s_4, s_5\}$. The resulting scheme can tolerate the failure of s_1, s_2, s_3 (a majority), but it cannot tolerate the failure of s_4, s_5 (a minority).

Quorum systems are static, but subsequent work proposed the notion of a *dynamic quorum system* [8, 13], which allows a user to add or remove elements to it, permitting the system to be reconfigured, while retaining the property that any two sets always intersect. Dynamic quorum systems by themselves are not sufficient to obtain reconfigurable storage, because one must populate storage nodes when the system is reconfigured and the scheme to populate is intertwined with the scheme to add or remove a node. Hence, reconfiguration algorithms such as those described here are needed.

Reconfiguration in state machine replication. State machine replication is a general technique to replicate a computation across several nodes. In a nutshell, one must ensure that all replicas start with the same state and see the same sequence of inputs (in the same order), thereby ensuring that they all undergo the same state transitions.

State machine replication and atomic storage are quite different abstractions. On one hand, the first is more general than the latter, because state machine replicas are not limited to reading and writing data: they may perform more complex read-modify-write operations or generate external events. On the other hand, it has been shown that state machine replication cannot be realized in asynchronous distributed systems subject to failures [10], whereas atomic storage can. Because of these inherent differences, the algorithms to implement these abstractions are quite different, and as a result the reconfiguration of state machines is quite different from the reconfiguration of atomic storage. However, some of the basic concepts we introduced apply to state machines as well, such as the requirements to obtain liveness in Section 3. For a tutorial on reconfiguration of state machines, see [16].

Group communication and virtual synchrony. Virtual synchrony [5] is an abstraction that comprises (1) a membership service that maintains a group of active nodes, and (2) a broadcast service for sending messages to the current members of the group, while providing reliable delivery and possibly total ordering of messages. Virtual synchrony can be used to implement general dynamic reliable services. This is done by using broadcast to deliver operations to service replicas, and by using the membership service to reconfigure the replica set. The methods we present share commonalities with the virtual synchrony methods, such as exposing reconfigurations and handling configuration changes via agreement. However, traditionally, the virtual synchrony approach pauses operation while a reconfiguration occurs, whereas we focus on *on-line* reconfiguration schemes.

9 Conclusion

Reconfigurable distributed storage systems rely on subtle techniques to ensure that reconfiguration operations maintain the expected semantics of the storage system. We explained two such techniques, one that relies on consensus, and another that does not. The first technique is simpler, but it requires a system where consensus can be implemented, whereas the second technique can be used in any system, even asynchronous ones. We hope that new techniques will be discovered that combine the benefits of both approaches. The liveness conditions of a reconfigurable storage system require a careful specification. We have explained how one can arrive at such specification. The specification we gave is weak and therefore general: we expect any reasonable reconfigurable system to satisfy it. However, we believe that stronger liveness specifications (i.e., specifications that ensure liveness in more cases) are attainable. We have also explained how to deal with the problems that arise when there are Byzantine failures. These techniques were developed for a system where consensus is available—hence they do not work in asynchronous systems. It remains to be seen whether reconfiguration is possible with Byzantine failures in asynchronous systems.

Acknowledgements. We are grateful to Panagiota Fatourou for comments that helped improve the presentation of this article.

References

[1] M. K. Aguilera and S. Frolund. Strict linearizability and the power of aborting. Technical Report HPL-2003-241, HP Labs, Nov. 2003.

[2] M. K. Aguilera, I. Keidar, D. Malkhi, and A. Shraer. Dynamic atomic storage without consensus. In *ACM Symposium on Principles of Distributed Computing*, pages 17–25, Aug. 2009.

[3] L. Alvisi, D. Malkhi, E. Pierce, M. K. Reiter, and R. N. Wright. Dynamic byzantine quorum systems. In *International Conference on Dependable Systems and Networks*, pages 283–292, June 2000.

[4] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, Jan. 1995.

[5] K. P. Birman and T. A. Joseph. Exploiting virtual synchrony in distributed systems. In *ACM Symposium on Operating Systems Principles*, pages 123–138, Nov. 1987.

[6] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, Nov. 2002.

[7] G. Chockler, S. Gilbert, V. Gramoli, P. M. Musial, and A. A. Shvartsman. Reconfigurable distributed storage for dynamic networks. *Journal of Parallel and Distributed Computing*, 69(1):100–116, Jan. 2009.

[8] D. Davcev and W. A. Burkhard. Consistency and recovery control for replicated files. In *ACM symposium on Operating systems principles*, pages 87–96, Dec. 1985.

[9] C. Dwork, N. A. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, Apr. 1988.

[10] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, Apr. 1985.

[11] S. Gilbert, N. A. Lynch, and A. A. Shvartsman. RAMBO: A robust, reconfigurable atomic memory service for dynamic networks. To appear in the *Distributed Computing* journal.

[12] S. Gilbert, N. A. Lynch, and A. A. Shvartsman. RAMBO II: Rapidly reconfigurable atomic memory for dynamic networks. In *International Conference on Dependable Systems and Networks*, pages 259–268, June 2003.

[13] M. Herlihy. Dynamic quorum adjustment for partitioned data. *ACM Transactions on Database Systems*, 12(2):170–194, June 1987.

[14] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Prog. Lang. Syst.*, 12(3):463–492, Jan. 1990.

[15] S. Jajodia and D. Mutchler. Dynamic voting algorithms for maintaining the consistency of a replicated database. *ACM Transactions on Database Systems*, 15(2):230–280, 1990.

[16] L. Lamport, D. Malkhi, and L. Zhou. Reconfiguring a state machine. *SIGACT News*, 41(1):63–73, Mar. 2010.

[17] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[18] N. A. Lynch and A. A. Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *International Symposium on Distributed Computing*, pages 173–190, Oct. 2002.

[19] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.

[20] J.-P. Martin and L. Alvisi. A framework for dynamic Byzantine storage. In *International Conference on Dependable Systems and Networks*, pages 325–334, June 2004.

[21] Y. Saito, S. Frolund, A. Veitch, A. Merchant, and S. Spence. FAB: building distributed enterprise disk arrays from commodity components. In *International conference on Architectural support for programming languages and operating systems*, pages 48–58, Oct. 2004.

[22] A. Shraer, J.-P. Martin, D. Malkhi, and I. Keidar. Data-centric reconfiguration with network-attached disks. In *ACM Workshop on Large Scale Distributed Systems and Middleware*, 2010.

[23] E. Yeger Lotem, I. Keidar, and D. Dolev. Dynamic voting for consistent primary components. In *ACM Symposium on Principles of Distributed Computing*, pages 63–71, Aug. 1997.

[24] L. Zhou, F. B. Schneider, and R. van Renesse. COCA: A secure distributed on-line certification authority. *ACM Transactions on Computer Systems*, 20(4):329–368, Nov. 2002.