Efficient and Modular Consensus-Free Reconfiguration for Fault-Tolerant Storage*†

Eduardo Alchieri[1], Alysson Bessani[2], Fabíola Greve[3], and Joni da Silva Fraga[4]

1 **University of Brasília, Brazil**
alchieri@unb.br
2 **LaSIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal**
anbessani@ciencias.ulisboa.pt
3 **Federal University of Bahia, Brazil**
fabiola@dcc.ufba.br
4 **Federal University of Santa Catarina, Brazil**
fraga@das.ufsc.br

Abstract

Quorum systems are useful tools for implementing consistent and available storage in the presence of failures. These systems usually comprise of a static set of servers that provide a fault-tolerant read/write register accessed by a set of clients. We consider a dynamic variant of these systems and propose FreeStore, a set of fault-tolerant protocols that emulates a register in dynamic asynchronous systems in which processes are able to join/leave the set of servers during the execution. These protocols use a new abstraction called *view generators*, that captures the agreement requirements of reconfiguration and can be implemented in different system models with different properties. Particularly interesting, we present a reconfiguration protocol that is *modular, efficient, consensus-free* and *loosely coupled* with read/write protocols. An analysis and an experimental evaluation show that the proposed protocols improve the overall system performance when compared with previous solutions.

1998 ACM Subject Classification C.2.4 Distributed Systems

Keywords and phrases Distributed Systems, Reconfiguration, Fault-Tolerant Quorum Systems

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2017.26

1 Introduction

Quorum systems [13] are a fundamental abstraction to ensure consistency and availability of data stored in replicated servers. Apart from their use as building blocks of synchronization protocols (e.g., consensus [7, 19]), quorum-based protocols for read/write (r/w) register implementation are appealing due to their scalability and fault tolerance: the r/w operations do not need to be executed in all servers, but only in a quorum of them. The consistency of the stored data is ensured by the intersection between any two quorums.

Quorum systems were initially studied in static environments, where servers are not allowed to join or leave the system during execution [4,13]. This approach is not adequate for

* This work was partially supported by CNPq (Brazil) through project FREESTORE (Universal 457272/2014-7) and by FCT (Portugal) through projects LaSIGE (UID/CEC/00408/2013) and IRCoC (PTDC/EEI-SCR/6970/2014).

† A full version of the paper is available at [3], https://arxiv.org/abs/1607.05344.

long lived systems since given a sufficient amount of time, there might be more faulty servers than the threshold tolerated, affecting the system correctness. Beyond that, this approach does not allow a system administrator to deploy new machines (to deal with increasing workloads) or replace old ones at runtime. Moreover, these protocols can not be used in many systems where, by their very nature, the set of processes that compose the system may change during its execution (e.g., MANETs and P2P overlays).

Reconfiguration is the process of changing the set of nodes that comprise the system. Previous solutions proposed for reconfigurable storage, by implementing dynamic quorum systems, rely on consensus for reconfigurations in a way that processes agree on the set of servers (view) supporting the storage [14, 20]. Although adequate, since the problem of changing views resembles an agreement problem, this approach is not the most efficient or appropriate. Besides the costs of running the protocol, consensus is known to not be solvable in asynchronous environments [11]. Moreover, atomic shared memory emulation can be implemented in static asynchronous systems without requiring consensus [4].

DynaStore [2] was the first protocol that implement dynamic atomic storage without relying on consensus for reconfigurations. These reconfigurations may occur at any time and generate a graph of views from which it is possible to identify a sequence of views in which clients need to execute their r/w operations (Figure 1, left). Unfortunately, DynaStore has two serious drawbacks: (1) its reconfiguration and r/w protocols are strongly tied and (2) its performance is significantly worse than consensus-based solutions in synchronous executions, which are the norm. The first issue is particularly important since it means that DynaStore r/w protocols (as well as other consensus-based works like RAMBO [14]) explicitly deal with reconfigurations and are quite different from static r/w protocols (e.g., ABD protocol [4]).

More recently, the SmartMerge [17] and SpSn [12] protocols improved DynaStore by separating the reconfiguration and r/w protocols. A common framework was proposed for their implementations [21]. Although these approaches make it easy to adapt other static register implementations to dynamic environments, they do not fully decouple the execution of r/w and reconfiguration protocols, since before they execute each r/w operation, it is necessary to access some reconfiguration abstraction to check for updates in the system. These abstractions are implemented by a set of static single-writer multi-reader (SWMR) registers. We show by experiments (Section 7) that this design decision significantly reduces the system performance, even in periods without reconfigurations.

In this paper we present FreeStore, a set of algorithms for implementing fault-tolerant atomic [18] and wait-free [15] storage that allows the reconfiguration of the set of servers at runtime. FreeStore is composed by two types of protocols: (1) r/w protocols and (2) the reconfiguration protocol. Read/write protocols can be adapted from static register implementations (e.g., the classical ABD [4], as used in this paper). The reconfiguration protocol – our main contribution – is used to change the set of replicas supporting the storage. The key innovation of the FreeStore reconfiguration protocol is the use of *view generators*, a new abstraction that captures the agreement requirements of reconfiguration protocols. We provide two implementations of view generators – based on consensus and consensus-free – and compare how efficiently they can be used to solve reconfiguration.

FreeStore improves the state of the art in at least three aspects: *modularity*, *efficiency* and *simplicity*. The modularity of the proposed protocols is twofold: it separates the reconfiguration from the r/w protocols, allowing other static storage protocols (e.g., [8, 10]) to be adapted to our dynamic model, and introduces the notion of view generators, capturing the agreement requirements of a reconfiguration protocol. Moreover, this modularity is designed in a way that it does not impact r/w operations in periods without reconfigurations. In

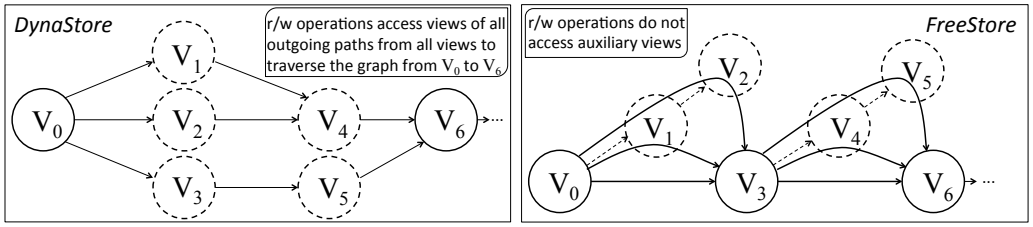

Figure 1 View convergence strategies of DynaStore [2] (left) and FreeStore (right). Dotted circles represent the auxiliary/non-established views that the system may experience during a reconfiguration. Solid circles represent the installed/established views process must converge to.

terms of performance, both the r/w and reconfiguration protocols of FreeStore require less communication steps than their counterparts from previous works, either consensus-based [14] or consensus-free [2,12,17]. In particular, FreeStore' consensus-free reconfiguration requires less communication steps than other consensus-based reconfiguration protocols in the best case, matching the intuition that a consensus-free approach should be faster than another that rely on consensus. Finally, the consensus-free variant of FreeStore introduces a novel reconfiguration strategy that reduces the number of intermediary installed views that a process must traverse to reach the current installed view with all requested updates (see Figure 1, right). This strategy is arguably easier to understand than the one used in previous works, shedding more light on the study of consensus-free reconfiguration protocols.

In summary, this paper makes the following contributions:

1. It introduces the notion of *view generators*, an abstraction that captures the agreement requirements of a storage reconfiguration protocol and provides two implementations: the consensus-based *perfect view generator* and the consensus-free *live view generator*.
2. It shows that *safe* and *live* dynamic fault-tolerant atomic storage can be implemented using the proposed view generators and discusses the tradeoffs between them.
3. It presents FreeStore, the first dynamic atomic storage system in which the reconfiguration protocol *(a)* can be configured to be consensus-free or consensus-based and *(b)* is fully decoupled from the r/w protocols, increasing the system performance and making it easy to adapt other static register implementation to dynamic environments.
4. It presents some experiments and a detailed comparison of FreeStore with previous protocols [2,12,14,17], showing that FreeStore is faster (i.e., requires less communication steps) than these protocols.

2 Preliminary Definitions

2.1 System Model

We consider a fully-connected distributed system composed by a universe of processes U, that can be divided in two non-overlapping subsets: an infinite set of servers $\Pi = \{1, 2, ...\}$; and an infinite set of clients $C = \{c_1, c_2, ...\}$. Clients access the storage system provided by a subset of the servers (a *view*) by executing read and write operations (*r/w* for short). Each process (client or server) of the system has a unique identifier. Servers and clients are prone to *crash failures*. Crashed processes are said to be *faulty*. A process that is not faulty is said to be *correct*. Moreover, there are *reliable channels* connecting all pairs of processes [6].

We assume an *asynchronous distributed system* in which there are no bounds on message transmission delays and processing times. However, each server has access to a local clock

used to trigger reconfigurations. These clocks are not synchronized and do not have any bounds on their drifts, being nothing more than counters that keep increasing. Besides that, there is a real-time clock not accessed by processes, used in definitions and proofs.

2.2 Dynamic Storage Properties

During a dynamic system execution, a sequence of views is installed to account for replicas joining and leaving. In the following we describe some preliminary definitions and the properties satisfied by FreeStore.

Updates. We define $update = \{+, -\} \times \Pi$, where the tuple $\langle +, i\rangle$ (resp. $\langle -, i\rangle$) indicates that server i asked to join (resp. leave) the system. A *reconfiguration* procedure takes into account updates to define a new system's configuration, which is represented by a *view*.

Views. A view v is composed by a set of *updates* (represented by $v.entries$) and its associated membership (represented by $v.members$). Consequently, $v.members = \{i \in \Pi \colon \langle +, i\rangle \in v.entries \wedge \langle -, i\rangle \notin v.entries\}$. To simplify the notation, we sometimes use $i \in v$ and $|v|$ meaning $i \in v.members$ and $|v.members|$, respectively. Notice that a server i can join and leave the system only once, but this condition can be relaxed in practice if we add an epoch number on each reconfiguration request.

We say a view v is *installed* in the system if some correct server $i \in v$ considers v as its *current view* and answers client r/w operations on this view. When v is installed, we say that the previous installed view (before v) was *uninstalled* from the system. At any time t, we define $V(t)$ to be the *most up-to-date view* (see definition below) installed in the system. We consider that $V(t)$ remains *active* from the time it is installed in the system until *all correct servers* of another most up-to-date view $V(t'), t' > t$, installs $V(t')$.

Comparing views. We compare two views v_1 and v_2 by comparing their *entries*. We use the notation $v_1 \subset v_2$ and $v_1 = v_2$ as an abbreviation for $v_1.entries \subset v_2.entries$ and $v_1.entries = v_2.entries$, respectively. If $v_1 \subset v_2$, then v_2 is *more up-to-date* than v_1.

Bootstrapping. We assume a non-empty initial view $V(0)$ known to all processes. At system startup, each server $i \in V(0)$ receives an initial view $v_0 = \{\langle +, j\rangle \colon j \in V(0)\}$.

Views vs. r/w operations. At any time t, r/w operations are executed only in $V(t)$. When a server i asks to *join* the system, these operations are disabled on it until an *enable operations* event occurs. After that, i remains able to process r/w operations until it asks to leave the system, which will happen after the occurence of a *disable operations* event.

▶ **Definition 1** (FreeStore properties)**.** FreeStore satisfies the following properties:

- **Storage Safety**: The r/w protocols satisfy the safety properties of an atomic r/w register [18].
- **Storage Liveness**: Every r/w operation executed by a correct client eventually completes.
- **Reconfiguration – Join Safety**: If a server j installs a view v such that $i \in v$, then server i has invoked the *join* operation or i is member of the initial view.

- **Reconfiguration – Leave Safety**: If a server j installs a view v such that $i \notin v \wedge (\exists v' : i \in v' \wedge v' \subset v)$, then server i has invoked the *leave* operation.[1]
- **Reconfiguration – Join Liveness**: Eventually, the *enable operations* event occurs at every correct server that has invoked a *join* operation.
- **Reconfiguration – Leave Liveness**: Eventually, the *disable operations* event occurs at every correct server that has invoked a *leave* operation.

2.3 Additional Assumptions for Dynamic Storage

Dynamic fault-tolerant storage protocols [2, 12, 14, 17, 20] require the following additional assumptions to deal with the dynamism.

▶ **Assumption 2** (Fault threshold). *For each view v, we denote $v.f$ as the number of faults tolerated in v and assume that $v.f \leq \lfloor \frac{|v.members|-1}{2} \rfloor$.*

▶ **Assumption 3** (Quorum size). *For each view v, we assume quorums of size $v.q = \lceil \frac{|v.members|+1}{2} \rceil$.*

These two assumptions are a direct adaptation of the optimal resilience for fault-tolerant quorum systems [4] to account for multiple views and only need to hold for views present in the generated view sequences (see Section 3).

▶ **Assumption 4** (Gentle leaves). *A correct server $i \in V(t)$ that asks to leave the system at time t remains in the system until it knows that a more up-to-date view $V(t'), t' > t, i \notin V(t')$ is installed in the system.*

This assumption ensures that a *correct* leaving server will participate in the reconfiguration protocol that installs the new view without itself, i.e., it cannot leave the system before a new view accounting for its removal is installed. Other dynamic systems require similar assumption: departing replicas need to stay available to transfer their state to arriving replicas. Notice the fault threshold accounts for faults while the view is being reconfigured.

▶ **Assumption 5** (Finite reconfigurations). *The number of updates requested in an execution is finite.*

As in other dynamic storage systems, this assumption is fundamental to ensure r/w operations termination. It ensures that a client will restart phases of a r/w operation a finite number of times, and thus, eventually complete its operation. In practice, updates could be infinite as long as each r/w is concurrent with a finite number of reconfigurations.

3 View Generators

View generators are distributed oracles used by servers to generate sequences of new views for system reconfiguration. This module aims to capture the agreement requirements of reconfiguration algorithms. In order to be general enough to be used for implementing consensus-free algorithms, such requirements are reflected in the sequence of generated views, and not directly on the views. This happens because, as described in previous works [1, 2, 12, 14, 17], the key issue with reconfiguration protocols is to ensure that the sequence of (possibly conflicting) views generated during a reconfiguration procedure will converge to a single view with all requested view updates.

[1] We can relax this property and adapt our protocol (Section 4) to allow that other process issues the leave operation on behalf of a crashed process.

For each view v, each server $i \in v$ associates a view generator $\mathcal{G}_i^v$ with v in order to generate a succeeding sequence of views. Server i interacts with a view generator through two primitives: (1) $\mathcal{G}_i^v.gen\_view(seq)$, called by i to propose a new view sequence seq to update v; and (2) $\mathcal{G}_i^v.new\_view(seq')$, a callback invoked by the view generator to inform i that a new view sequence seq' was generated for succeeding v. An important remark about this interface is that there is no one-to-one relationship between *gen_view* and *new_view*: a server i may not call the first but receive several upcalls on the latter for updating the *same view* (e.g., due to reconfigurations started by other servers). However, if i calls $\mathcal{G}_i^v.gen\_view(seq)$, it will eventually receive at least one upcall to $\mathcal{G}_i^v.new\_view(seq')$.

Similarly to other distributed oracles (e.g., failure detectors [7]), view generators can implement these operations in different ways, according to the different environments they are designed to operate (e.g., synchronous or asynchronous systems). However, in this paper we consider view generators satisfying the following properties.

▶ **Definition 6** (View Generators)**.** A generator $\mathcal{G}_i^v$ (associated with v in server i) satisfy the following properties:

- **Accuracy**: we consider two variants of this property:
  - **Strong Accuracy**: for any $i, j \in v$, if i receives an upcall $\mathcal{G}_i^v.new\_view(seq_i)$ and j receives an upcall $\mathcal{G}_j^v.new\_view(seq_j)$, then $seq_i = seq_j$.
  - **Weak Accuracy**: for any $i, j \in v$, if i receives an upcall $\mathcal{G}_i^v.new\_view(seq_i)$ and j receives an upcall $\mathcal{G}_j^v.new\_view(seq_j)$, then either $seq_i \subseteq seq_j$ or $seq_j \subset seq_i$.
- **Non-triviality**: for any upcall $\mathcal{G}_i^v.new\_view(seq_i)$, $\forall w \in seq_i$, $v \subset w$.
- **Termination**: if a correct server $i \in v$ calls $\mathcal{G}_i^v.gen\_view(seq)$, then eventually it will receive an upcall $\mathcal{G}_i^v.new\_view(seq_i)$.

Accuracy and Non-triviality are safety properties while Termination is related to the liveness of view generation. Furthermore, the Non-triviality property ensures that generated sequences contain only updated views.

Using the two variants of accuracy we can define two types of view generators: $\mathcal{P}$, the *perfect view generator*, that satisfies *Strong Accuracy* and $\mathcal{L}$, the *live view generator*, that only satisfies *Weak Accuracy*. Our implementation of $\mathcal{P}$ requires consensus, while $\mathcal{L}$ can be implemented without such strong synchronization primitive.

3.1 Perfect View Generators – $\mathcal{P}$

Perfect view generators ensure that a single sequence of views is generated at all servers using the generators. Our implementation for $\mathcal{P}$ (Algorithm 1) uses a deterministic Paxos-like consensus protocol [19] that assumes a partially synchronous system model [9]. Under the Paxos framework, any server of v can be a proposer (calling $Paxos^v.propose(seq)$), but all servers of v are acceptors and learners (they receive an upcall $Paxos^v.learn(seq')$, even if they did not propose anything). We also assume that once a value is learned for a Paxos instance associated with v, this value is locked and no other value will be learned. Notice that a server only proposes a sequence if the new views are strict extensions of v, ensuring Non-triviality. Termination and Strong Accuracy properties comes directly from the Agreement and Termination properties of the underlying consensus algorithm [7, 19].

3.2 Live View Generators – $\mathcal{L}$

Algorithm 2 presents an implementation for *Live view generators* ($\mathcal{L}$). Our algorithm does not require a consensus building block, being thus implementable in asynchronous systems.

Algorithm 1 $\mathcal{P}$ associated with v - server $i \in v$.

upon $\mathcal{G}_i^v.gen\_view(seq)$
1) **if** $\forall w \in seq : v \subset w$ **then** $Paxos^v.propose(seq)$ //starts a consensus in v
upon $Paxos^v.learn(seq')$ // when decides by seq' ...
2) $\mathcal{G}_i^v.new\_view(seq')$ // ... inform this to process.

Algorithm 2 $\mathcal{L}$ associated with v - server $i \in v$.

functions: Auxiliary functions
$most\_updated(seq) \equiv w$: $(w \in seq) \wedge (\nexists w' \in seq$: $w \subset w')$
variables: Sets used in the protocol
$\texttt{SEQ}^v \leftarrow \emptyset$ // proposed view sequence
$\texttt{LCSEQ}^v \leftarrow \emptyset$ // last converged view sequence known
procedure $\mathcal{G}_i^v.gen\_view(seq)$
1) **if** $\texttt{SEQ}^v = \emptyset \wedge \forall w \in seq : v \subset w$ **then**
2) $\texttt{SEQ}^v \leftarrow seq$
3) $\forall j \in v, send\langle$SEQ-VIEW$, \texttt{SEQ}^v\rangle$ to j
upon receipt of $\langle$SEQ-VIEW$, seq\rangle$ from j
4) **if** $\exists w \in seq$: $w \notin \texttt{SEQ}^v$ **then**
5) **if** $\exists w, w'$: $w \in seq \wedge w' \in \texttt{SEQ}^v \wedge w \not\subset w' \wedge w' \not\subset w$ **then**
6) $w \leftarrow most\_updated(\texttt{SEQ}^v)$
7) $w' \leftarrow most\_updated(seq)$
8) $\texttt{SEQ}^v \leftarrow \texttt{LCSEQ}^v \cup \{w.entries \cup w'.entries\}$
9) **else**
10) $\texttt{SEQ}^v \leftarrow \texttt{SEQ}^v \cup seq$
11) $\forall k \in v, send\langle$SEQ-VIEW$, \texttt{SEQ}^v\rangle$ to k
upon receipt of $\langle$SEQ-VIEW$, \texttt{SEQ}^v\rangle$ from $v.q$ servers in v
12) $\texttt{LCSEQ}^v \leftarrow \texttt{SEQ}^v$
13) $\forall k \in v, send\langle$SEQ-CONV$, \texttt{SEQ}^v\rangle$ to k
upon receipt of $\langle$SEQ-CONV$, seq'\rangle$ from $v.q$ servers in v
14) $\mathcal{G}_i^v.new\_view(seq')$

On the other hand, it can generate different sequences in different servers for updating the same view. We bound such divergence by exploiting Assumptions 2 and 3, which ensure that any quorum of the system will intersect in at least one correct server, making any generated sequence for updating v be contained in any other posterior sequence generated for v (Weak Accuracy). Furthermore, the servers keep updating their proposals until a quorum of them converges to a sequence containing all proposed views (or combinations of them), possibly generating some intermediate sequences before this convergence.

To generate a new sequence of views, a server $i \in v$ uses an auxiliary function *most_updated* to get the most up-to-date view in a sequence of views (i.e., the view that is not contained in any other view of the sequence). Moreover, each server keeps two local variables: $\texttt{SEQ}^v$ – the last view sequence proposed by the server – and $\texttt{LCSEQ}^v$ – the last sequence this server converged. When server $i \in v$ starts its view generator, it first verifies (1) if it already made a proposal for updating v and (2) if the sequence being proposed contains only updated views (line 1). If these two conditions are met, it sends its proposal to the servers in v (lines 2-3).

Different servers of v may propose sequences containing different views and therefore these views need to be organized in a sequence. When a server $i \in v$ receives a proposal for a view sequence from $j \in v$, it verifies (line 4) if this proposal contains some view it did not know yet (notice a server could receive this message even if its view generator was not yet initialized, i.e., $\texttt{SEQ}^v = \emptyset$). If this is the case, i updates its proposal ($\texttt{SEQ}^v$) according to two mutually exclusive cases:

- Case 1 [There are *conflicting views* in the sequence proposed by i and the sequence received from j (lines 5-8)]: In this case i creates a new sequence containing the last converged sequence ($\texttt{LCSEQ}^v$) and a new view with the union of the two most up-to-date conflicting views. This maintains the containment relationship between any two generated view sequences.
- Case 2 [The sequence proposed by i and the received sequence *can be composed in a new sequence* (lines 9-10)]: The new sequence is the union of the two known sequences.

In both cases, a new proposal containing the new sequence is disseminated (line 11). When i receives the same proposal from a quorum of servers in v, it *converges* to $\texttt{SEQ}^v$ and stores it in $\texttt{LCSEQ}^v$, informing other servers of v about it (lines 12-13). When i knows that a quorum of servers of v converged to some sequence seq', it *generates* seq' (line 14).

Correctness (full proof in [3]). The algorithm ensures that if a quorum of servers converged to a sequence seq' (lines 12-13), then *(1)* such sequence will be generated (line 14) and *(2)* any posterior sequence generated will contain seq' (lines 8 and 10), ensuring Weak Accuracy. This holds due to the quorum intersection: at least one correct server needs to participate in the generation of both sequences and this server applies the rules in Cases 1 and 2 to ensure that sequences satisfy the containment relationship. The Termination property is ensured by the fact that *(1)* each server makes at most one initial proposal (lines 1-3); *(2)* servers keep updating their proposals until a quorum agree on some proposal; and *(3)* there is always a quorum of correct servers in v.

4 FreeStore Reconfiguration

A server running FreeStore reconfiguration algorithm uses a view generator associated with its current view cv to process one or more reconfiguration requests (joins and leaves) that will lead the system from cv to a new view w. Algorithm 3 describes how a server i executes reconfigurations. In the following sections we first describe how view generators are started and then we proceed to discuss the behavior of this algorithm when started with either $\mathcal{L}$ (Section 4.2) or $\mathcal{P}$ (Section 4.3).

4.1 View Generator Initialization

Algorithm 3 describes how a server i processes reconfiguration requests and starts a view generator associated with its current view cv (lines 1-7). A server j that wants to join the system needs first to find the current view cv and then to execute the *join* operation (lines 1-2), sending a tuple $\langle +, j \rangle$ to the members of cv. Servers leaving the system do something similar, through the *leave* operation (lines 3-4). When i receives a reconfiguration request from j, it verifies if the requesting server is using the same view as itself; if this is not the case, i replies its current view to j (omitted from the algorithm for brevity). If they are using the same view and i did not execute the requested reconfiguration before, it stores this request in its set of pending updates `RECV` and sends an acknowledgment to j (lines 5-6).

For the sake of performance, a local *timer* has been defined in order to periodically process the updates requested in a view, that is, the next system reconfiguration. A server $i \in cv$ starts a reconfiguration for cv when its timer expires and i has some pending reconfiguration requests (otherwise, the timer is renewed). The view generator is started with a sequence containing a single view representing the current view plus the pending updates (line 7).

Algorithm 3 FREESTORE reconfiguration - server i.

functions: Auxiliary functions
  $least\_updated(seq) \equiv w\colon (w \in seq) \wedge (\nexists w' \in seq\colon w' \subset w)$
variables: Sets used in the protocol
  $cv \leftarrow v_0$ // the system current view known by i
  $\texttt{RECV} \leftarrow \emptyset$ // set of received updates
procedure *join*()
1) $\forall j \in cv, send\langle$RECONFIG$, \langle +, i\rangle, cv\rangle$ to j
2) **wait** for $\langle$REC-CONFIRM$\rangle$ replies from $cv.q$ servers in cv
procedure *leave*()
3) $\forall j \in cv, send\langle$RECONFIG$, \langle -, i\rangle, cv\rangle$ to j
4) **wait** for $\langle$REC-CONFIRM$\rangle$ replies from $cv.q$ servers in cv
upon receipt of $\langle$RECONFIG$, \langle *, j\rangle, cv\rangle$ from j and $\langle *, j\rangle \notin cv$
5) $\texttt{RECV} \leftarrow \texttt{RECV} \cup \{\langle *, j\rangle\}$
6) $send\langle$REC-CONFIRM$\rangle$ to j
upon ($timeout$ for cv) $\wedge(\texttt{RECV} \neq \emptyset)$
7) $\mathcal{G}_i^{cv}.gen\_view(\{cv \cup \texttt{RECV}\})$
upon $\mathcal{G}_i^{ov}.new\_view(seq)$ //$\mathcal{G}$ generates a new sequence of views to update ov (usually $ov = cv$)
8) $w \leftarrow least\_updated(seq)$ //the next view in the sequence seq
9) $R\text{-}multicast(\{j\colon j \in ov \vee j \in w\}, \langle$INSTALL-SEQ$, w, seq, ov\rangle)$
upon $R\text{-}delivery(\{j\colon j \in ov \vee j \in w\}, \langle$INSTALL-SEQ$, w, seq, ov\rangle)$
10) **if** $i \in ov$ **then** //i is member of the previous view in the sequence
11) **if** $cv \subset w$ **then** *stop the execution of r/w operations* //if w is more up-to-date than cv stop r/w
12) $\forall j \in w, send\langle$STATE-UPDATE$, \langle val, ts\rangle, \texttt{RECV}\rangle$ to j //i sends its state to servers in the next view
13) **if** $cv \subset w$ **then** //w is more up-to-date than cv and the system will be reconfigured from cv to w
14) **if** $i \in w$ **then** //if i is in the new view...
15) **wait** for $\langle$STATE-UPDATE$, *, *\rangle$ messages from $ov.q$ servers in ov //... it updates...
16) $\langle val, ts\rangle \leftarrow \langle val_h, ts_h\rangle$, pair with highest timestamp among the ones received //... its state...
17) $\texttt{RECV} \leftarrow \texttt{RECV} \cup \{$update requests from STATE-UPDATE messages$\} \setminus w.entries$
18) $cv \leftarrow w$ //... and its current view to w
19) **if** $i \notin ov$ **then** *enable operations* //i is joining the system
20) $\forall j \in ov \setminus cv, send\langle$VIEW-UPDATED$, cv\rangle$ to j //inform servers in $ov \setminus cv$ that they can leave
21) **if** $(\exists w' \in seq\colon cv \subset w')$ **then** //there are views more up-to-date than cv in seq...
22) $seq' \leftarrow \{w' \in seq\colon cv \subset w'\}$ //... gather these views...
23) $\mathcal{G}_i^{cv}.gen\_view(seq')$ //... and propose them (going back to Algorithm 2)
24) **else**
25) *resume the execution of r/w operations* in $cv = w$ and start a timer for cv //w is installed
26) **else** //i is leaving the system
27) *disable operations*
28) **wait** for $\langle$VIEW-UPDATED$, w\rangle$ messages from $w.q$ servers in w and then **halt**

4.2 Reconfiguration using $\mathcal{L}$

Overview. Given a sequence $seq : v_1 \rightarrow \ldots \rightarrow v_k \rightarrow w$ generated by *live view generators* ($\mathcal{L}$) for updating a view v, Algorithm 3 ensures that only the last view w will be installed in the system. The other k *auxiliary views* are used only as intermediate steps for installing w.

The use of auxiliary views is fundamental to ensure that no write operation executed in any of the views of seq (in case they are installed in some server) "is lost" by the reconfiguration processing. This is done through the execution of a *"chain of reads"* in which servers of v transfer their state to servers of v_1, which transfer their state to servers of v_2 and so on until servers of w have the most up-to-date state. To avoid consistency problems, r/w operations are disabled during each of these state transfers.

It is important to remark that since we do not use consensus, a subsequence $seq' : v_1 \rightarrow \ldots \rightarrow v_j, j \leq k$, of seq may lead to the installation of v_j in some servers that did not know seq and that these servers may execute r/w operations in this view. However, the algorithm ensures these servers eventually will reconfigure from v_j to the most up-to-date view w.

Protocol. Algorithm 3 (lines 8-28) presents the core of the FreeStore reconfiguration protocol. This algorithm uses an auxiliary function *least_updated* to obtain the least updated view in a sequence of views (i.e., the one that is contained in all other views of the sequence) and two local variables: the aforementioned `RECV` – used to store pending reconfiguration requests – and cv – the current view of the server (initially v_0).

When the view generator associated with some view ov (we use ov instead of cv because view generators associated with *old views*, $ov \subseteq cv$, still active can generate new sequences) reports the generation of a sequence of views seq, the server obtains the least updated view w of seq and proposes this sequence for updating ov through an INSTALL-SEQ message sent to the servers of both, ov and w. Once view generators could generate different sequences at different processes, we employ a *reliable multicast* primitive [7] to ensure all correct servers in ov and w process seq (lines 8-9). This primitive can be efficiently implemented in asynchronous systems with a message retransmission at the receivers before its delivery [7].

The current view is updated through the execution of lines 10-28. First, if the server is a member of the view being updated ov, it must send its state (usually, the register's value and timestamp) to the servers in the new view to be installed (lines 10-12). However, if the server will be updating its current view (i.e., if w is more up-to-date than cv) it first needs to stop executing client r/w operations (line 11) and enqueue these operations to be executed when the most up-to-date view in the sequence is installed (line 25, as discussed below). A server will update its current view only if the least updated view w of the proposed sequence is more up-to-date than its current view cv (line 13). If this is the case, either (1) server i will be in the next view (lines 14-25) or (2) not (line 26-28).

- Case 1 [i will be in the next view (it may be joining the system)]: If the server will be in the next view w, it first waits for the state from a quorum of servers from the previous view ov and then defines the current value and timestamp of the register (lines 15-16), similarly to what is done in the 1st phase of a read operation (see Section 5). After ensuring that its state is updated, the server updates cv to w and, if it is joining the system, it enables the processing of r/w operations (which will be queued until line 25 is executed). Furthermore, the server informs leaving servers that its current view was updated (line 20). The final step of the reconfiguration procedure is the verification if the new view will be installed or not (in case it is an auxiliary view). If $cv = w$ is not the most up-to-date view of the generated sequence seq, a new sequence with more up-to-date views than cv will be proposed for updating it (lines 21-23). Otherwise, cv is installed and server i resumes processing r/w operations (lines 24-25).
- Case 2 [i is leaving the system]: A server leaving the system only halts after ensuring that the view w to which it sent its state was started in a quorum of servers (lines 27-28).

Although the algorithm restarts itself in line 23, it eventually terminates since the number of reconfiguration requests is finite (Assumption 5). Furthermore, since all sequences generated by $\mathcal{L}$ can be composed in an unique sequence, when the reconfiguration terminates in all correct servers, they will have installed the same view with all requested updates.

Correctness (full proof in [3]). Algorithm 3 ensures that *an unique sequence of views is installed in the system* due to the following: (1) if a view w is installed, any previously installed view $w' \subset w$ is uninstalled and will not be installed anymore (lines 11, 18 and 25); consequently, (2) no view more up-to-date than w is installed and the installed views form an unique sequence. By Assumption 5, the reconfiguration procedure always terminate by installing a final view v_{final}. The *Storage Safety* and *Storage Liveness* properties are discussed

in Section 5. The remaining properties of Definition 1 are ensured as follows. *Reconfiguration Join/Leave Safety* are trivially ensured by the fact that only a server i sends the update request $\langle +, i\rangle/\langle -, i\rangle$ (lines 1-4). *Reconfiguration Join/Leave Liveness* are ensured by the fact that if an update request from a server i is stored in `RECV` of a quorum, then it is processed in the next reconfiguration since a quorum with the same proposal is required to generate a view sequence (Algorithm 2). Moreover, update requests received during a reconfiguration are sent to the next view (lines 12-17).

4.3 Reconfiguration using $\mathcal{P}$

If $\mathcal{P}$ is used with the FreeStore reconfiguration protocol (Algorithm 3), all generators will generate the same sequence of views (Strong Accuracy) with a single view w. This will lead the system directly from its current view cv to w (lines 22-23 will never be executed).

5 Read and Write Protocols

This section discusses how a static storage protocol can be adapted to dynamic systems by using FreeStore reconfigurations. Since reconfigurations are decoupled from r/w protocols, they are very similar to their static versions. In a nutshell, there are two main requirements for using our reconfiguration protocol. First, each process (client or server) needs to handle a current view variable cv that stores the most up-to-date view it knows. All r/w protocol messages carry cv and clients update it as soon as they discover that there is a more recent view installed in the system. The servers reject any operation issued to an old view, and reply their current view to the issuing client, which updates its cv. The client restarts the phase of the operation it is executing if it receives an updated view. The second requirement is that, before accessing the system, a client must obtain the system's current view. This can be done by making servers put the current view in a directory service [1] or making the client flood the network asking for it. Notice this is an intrinsic problem for any dynamic system and similar assumptions are required in previous reconfiguration protocols [2, 12, 14, 17, 20].

In this paper we extend the classical ABD algorithm [4] for supporting multiple writers and to work with the FreeStore reconfiguration. In the following we highlight the main aspects of these protocols (complete algorithms are found in [3]). The protocols to read and write from the dynamic distributed storage work in phases. Each phase corresponds to an access to a quorum of servers in cv. The *read protocol* works as follows:

- 1st Phase: a reader client requests a set of tuples $\langle val, ts\rangle$ from a quorum of servers in cv (val is the value the server stores and ts is its associated timestamp) and selects the one with highest timestamp $\langle val_h, ts_h\rangle$; the operation ends and returns val_h if all returned pairs are equal, which happens in executions without write contention or failures;
- 2nd Phase: otherwise, the reader client performs an additional *write-back phase* in the system and waits for confirmations from a quorum of servers in cv before returning val_h.

The *write protocol* works in a similar way:

- 1st Phase: a writer client obtains a set of timestamps from a quorum of servers in cv and chooses the highest, ts_h; the timestamp to be written ts is defined by incrementing ts_h and concatenating the writer id in its lowest bits;
- 2nd Phase: the writer sends a tuple $\langle val, ts\rangle$ to the servers of cv, writing val with timestamp ts, and waits for confirmations from a quorum.

The proposed decoupling of r/w protocols from reconfigurations (1) makes it easy to adapt other static fault-tolerant register implementations to dynamic environments (as long as they work in phases), (2) makes it possible to run r/w in parallel with reconfigurations,

and (3) does not impact the performance of a r/w in periods without reconfigurations. This happens because, differently from previous approaches [2, 12, 14, 17] where a r/w may access multiple views, in FreeStore a client executes these operations only in the most up-to-date installed view, which is received directly from the servers. To enable this, it is required that r/w operations to be blocked during the state transfer between views.

Correctness (full proof in [3]). The above algorithms implement atomic storage in the absence of reconfigurations [4]. When integrated with FreeStore, they also have to satisfy the *Storage Safety* and *Storage Liveness* properties of Definition 1. We start by discussing *Storage Liveness*, which follows directly from the termination of reconfigurations. As discussed before, clients' r/w operations are concluded only if they access a quorum of servers using the same view as the client, otherwise they are restarted. By Assumption 5, the system reconfiguration always terminate by installing some final view. Consequently, a client will restart phases of its operations a finite number of times until this final view is installed in a quorum. *Storage Safety* comes directly from three facts: (1) a r/w operation can only be executed in a single *installed* view (i.e., all servers in the quorum of the operation have the same installed current view), (2) all installed views form a unique sequence , and (3) any operation executed in a view v will still be "in effect" when a more up-to-date view w is installed. More precisely, assume $\langle val, ts \rangle$ is the last value read or written in v, and thus it was stored in $v.q$ servers from v. During a reconfiguration from v to w, r/w operations are disabled until all servers of v send $\langle val, ts \rangle$ to the servers of w (line 12), which terminate the reconfiguration only after receiving the state from a quorum of servers of v. Consequently, all servers who reconfigure to w will have $\langle val, ts \rangle$ as its register's state (lines 15-16). This ensures that any operation executed in w will not miss the operations executed in v.

6 Discussion

6.1 DynaStore vs. FreeStore

This section discusses some differences between DynaStore [2] and FreeStore. Although our focus is on comparing our approach with DynaStore, we also comment the relationship between these protocols and SpSn [12] and SmartMerge [17].

Convergence Strategy. In DynaStore, the reconfiguration process generate a graph of views through which it is possible to identify a sequence of *established* views (see Figure 1, left). A view that is not established works as an *auxiliary view*, that must be accessed during r/w operations. For any established view v, the maximum number of views that can immediately succeed it is $|v|$, and new views representing the combinations of these views could also be generated. In contrast, reconfigurations in FreeStore install only a single sequence of views (see Figure 1, right). In this case, different generated sequences of views are organized in an unique sequence of installed views. For each installed view v, our implementation of $\mathcal{L}$ bounds the number of generated view sequences to $|v| - v.q + 1$. SpSn and SmartMerge also install a sequence of ordered configurations (views).

Liveness. In DynaStore, a process executes a leave and halts the system. However, for any time t, there is a bound on the number of processes that can leave the system without compromising liveness. Let $F(t)$ be the set of processes that crashed until time t and $J(t)$ (resp. $L(t)$) the set of pending joins (resp. leaves) at t. The liveness condition of DynaStore states that fewer than $|V(t)|/2$ processes out of $V(t) \cup J(t)$ should be in $F(t) \cup L(t)$ [2]. In

Table 1 Communication steps of r/w operations.

Operation	*Updated View*				*Outdated View*			
	DS	SM	SpSn	FS	DS	SM	SpSn	FS
Read	12	8	14	2/4	19	16	22	4/6
Write	12	8	18	4	19	16	26	6

contrast, in FreeStore a process that executes a leave should wait for the installation of the updated view (without itself). If this restriction is not respected, the leaving server is considered faulty. This approach specifies a bound on the number of leaves for FreeStore: fewer than $|V(t)|/2$ processes out of $V(t)$ should be in $F(t) \cup L(t)$. SpSn does not specify a liveness condition and SmartMerge uses policies to prevent the generation of unsafe views, which in practice restricts the number of allowed leaves.

Normal Case Execution. In DynaStore, reconfigurations generate a graph of views through which it is possible to identify a sequence of *established* views. A view that is not established works as an *auxiliary view*, but must be accessed during r/w operations. For each view (established or not), DynaStore associates a *weak snapshot object* that is used to store updates. For any view v, its weak snapshot object wso_v is supported by a set S_v of $|v|$ static SWMR registers, i.e., one register for each member of v. During a r/w on v, wso_v must be accessed twice to verify if some update was executed on v. Each access to wso_v comprises two reads in each register of S_v. Thus, to execute a r/w on v, it is necessary a total of $4|v|$ (4 sequential, $|v|$ parallel) quorum accesses, with two communication steps each. SpSn and SmartMerge use a similar approach: before executing each r/w, a set of $|v|$ SWMR registers must be accessed to check for updates. In contrast, the overhead introduced by FreeStore on a r/w is the local verification if the client view is equal to the current view of the servers.

R/W and Reconfiguration Concurrency. A r/w operation needs to traverse the graph of views generated by DynaStore to find a view where it is safe to be executed (the most up-to-date established view). During the transversal, each edge of the graph must be accessed in order to verify if there is a more up-to-date view. For each view v, it is necessary to access its weak snapshot object, which requires $2|v|$ (2 sequential, $|v|$ parallel) quorum accesses. While the system is converging to some established view, the r/w operation does not terminate. Similarly, in SpSn and SmartMerge, a r/w concurrent with a reconfiguration also needs to access intermediary views to find the most up-to-date installed view. FreeStore follows a different approach that ensures r/w operations are directed to the most up-to-date installed view v, without accessing any auxiliary view. However, after some sequence of views for updating v is obtained, the r/w operations are stopped and can only terminate after the installation of the most up-to-date view of this sequence. Therefore, in this sense our approach resembles view-synchronous group communication [5, 6].

6.2 Performance of Read/Write Operations

Table 1 shows the number of communication steps demanded to execute a r/w operation with DynaStore (DS) [2], SmartMerge (SM) [17], SpSn [12] and FreeStore (FS), for a process that handles an updated or outdated view. A r/w operation in FreeStore requires at most a third of the number of communication steps required to execute it in DynaStore or SpSn and at most half of the steps required in SmartMerge. More important, it matches the performance of static protocols in the absence of reconfigurations.

Table 2 Communication steps for reconfiguration.

Consensus-free	*Best Case*	*Worst Case*	*Consensus-based*	*Best Case*	*Worst Case*
DynaStore [2]	23	$18\|v\| + 5$	RAMBO [14]	7	7
SmartMerge [17]	11	$6\|v\| + 5$	FreeStore ($\mathcal{P}$)	5	5
SpSn [12]	14	$8\|v\| + 6$			
FreeStore ($\mathcal{L}$)	4	$7\|v\| - 2v.q - 1$			

6.3 Consensus-free vs. Consensus-based Reconfiguration

Table 2 shows the number of communication steps required to process a reconfiguration using consensus-free (DynaStore, SmartMerge, SpSn and FreeStore with $\mathcal{L}$) and consensus-based (RAMBO and FreeStore with $\mathcal{P}$) algorithms. We present the number of communication steps for the best case scenario – when all processes propose the same updates in a reconfiguration – and for the worst case – when each process proposes different updates. In order to simplify our analysis, we consider that no reconfiguration is started concurrently with the one we are analyzing. Similarly, we assume a synchronous execution of the Paxos protocol [19], which requires only three communication steps, for consensus-based algorithms.

FreeStore reconfiguration is significantly more efficient than previous consensus-based and consensus-free protocols. FreeStore with $\mathcal{P}$ requires two less communication steps than RAMBO and FreeStore with $\mathcal{L}$ outperforms other consensus-free approaches by almost an order of magnitude. In particular, FreeStore with $\mathcal{L}$ presents the best performance among all considered reconfiguration protocols in the best case, which is expected to be the norm in practice. An open question is if this number constitutes a lower bound.

7 Experimental Evaluation

In this section we present an experimental evaluation of FreeStore (1) to quantify its overhead when compared with the (static) ABD protocol in periods without reconfigurations, and (2) to assess the negative impact of a reconfiguration in the system performance.

We implemented prototypes of the ABD [4], FreeStore and DynaStore [2] protocols in the *Go* programming language and conducted two sets of experiments in Emulab [22]: a micro-benchmark designed to evaluate the read and write throughput and latency in absence of reconfiguration; and an execution showing the performance of dynamic algorithms during faults and reconfigurations. We chose DynaStore to represent existing consensus-free dynamic protocols [2, 12, 17] to show that design decisions such as *checking a set of SWMR static registers to verify if some reconfiguration occurred before executing each r/w* and *coupling the execution of r/w and reconfigurations* have a significant impact in the system performance. Although comparing with DynaStore suffice for these goals, the interested reader can find a more extensive experimental evaluation of reconfiguration protocols in [16].

Experimental Setup. We used 18 *pc3000* (3.0 GHz 64-bit Pentium Xeon, 2GB of RAM and gigabit network cards) connected by a 100Mb switched network. We run each server in a separated machine, while the clients were uniformly distributed among the remaining machines. The software installed was Fedora 15 64-bit with *kernel* 2.6.20 and *Go* 1.2.

Micro-benchmarks. We start by reporting the latency and throughput results for the system configured with 3 servers (Figure 2). We used up to 18 clients to read or write a value of 512 bytes. Both latency and throughput was measured at the clients: the latency is the mean time demanded to perform a r/w operation discarding the 5% values with greater variance; the throughput is the sum of all client operations completed in a interval.

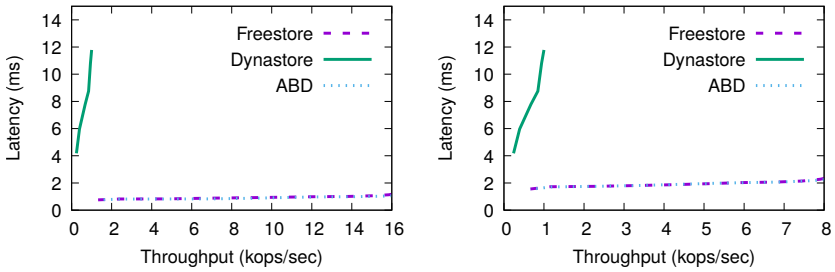

Figure 2 Latency vs. throughput for read (left) and write (right) operations; $n = 3$ and $f = 1$.

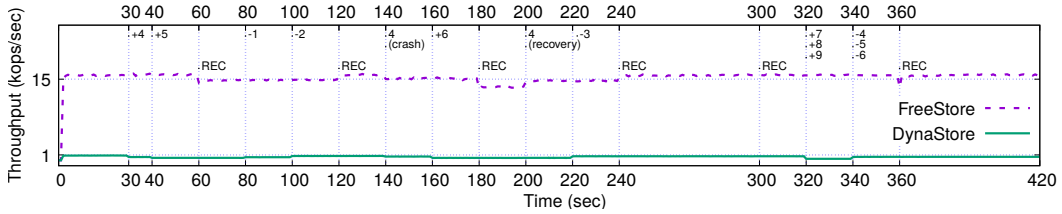

Figure 3 Throughput evolution across faults and reconfigurations for FreeStore and DynaStore.

These experiments show that FreeStore imposes a negligible overhead to the static ABD protocols. This happens because these protocols are very similar: the difference is that FreeStore messages must carry the current view to check if a client is using an updated view (we used hashes of the views; the complete views were used only when necessary an update in the client view). In contrast, DynaStore performs poorly because it must access a set of static SWMR registers to check for view updates before executing each r/w operation.

Reconfigurations and faults. This experiment considers the behavior of FreeStore with $\mathcal{L}$ (the result is similar with $\mathcal{P}$) and DynaStore protocols under reconfigurations, failures and recoveries. We observed how the throughput of these systems evolve over several events when a demanding workload is applied. We used an initial view with 3 servers ($v_0 = \{1, 2, 3\}$) and 18 clients that keep reading a value of 512 bytes over the course of 420 seconds. The results in Figure 3 show that FreeStore significantly outperforms DynaStore.

In the experiment, servers 4-9 asked to join at times 30, 40, 160, and 320 (servers 7-9), respectively, while servers 1-6 asked to leave at times 80, 100, 220, 340 (servers 4-6), respectively. DynaStore reconfigurations occurred each time an update was requested [2], while FreeStore reconfigurations were configured to occur periodically at each 60 seconds (notice that at time 360, a FreeStore reconfiguration replaces all servers in the system). Moreover, server 4 crashed and recovered at times 140 and 200, respectively.

This experiment shed light in how reconfigurations impact the performance of concurrent r/w operations. The mean time required for a FreeStore reconfiguration was 19 ms, with r/w operations blocked for only 4 ms (see Algorithm 3). Increasing the size of the value stored in the system may lead this time to increase. However, in all previous works on asynchronous reconfigurations [2, 12, 17], the state transfer happens during a r/w operation and, consequently, the time to finish these operations will also increase.

8 Conclusions

This paper presented a new approach to reconfigure fault-tolerant storage systems, which clarifies the differences between relying or not on consensus for agreement in the next view to be installed. The main result is a protocol that is simpler and cheaper (in terms of communication steps for either r/w operations or reconfigurations) than the previously proposed solutions. Furthermore, our approach fully decouples the execution of r/w operations and reconfigurations, imposing a negligible overhead to the static ABD protocol. Another interesting result is that, in the best case, FreeStore consensus-free reconfiguration is faster than protocols based on consensus.

A final contribution of this work is the introduction of a new abstraction called view generator. We believe that exploring different instantiations of this abstraction and their properties is an important avenue for future work.

References

1 Marcos K. Aguilera, Idit Keidar, Dahlia Malkhi, Jean-Philippe Martin, and Alexander Shraer. Reconfiguring replicated atomic storage: A tutorial. *Bulletin of EATCS: The Distributed Computing Column*, 2010.

2 Marcos K. Aguilera, Idit Keidar, Dahlia Malkhi, and Alexander Shraer. Dynamic atomic storage without consensus. *Journal of the ACM*, 58:7:1–7:32, 2011.

3 Eduardo Alchieri, Alysson Bessani, Fabiola Greve, and Joni Fraga. Efficient and modular consensus-free reconfiguration for fault-tolerant storage. *ArXiv*, 2016. `arXiv:1607.05344`.

4 Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, 1995.

5 Kenneth Birman and Thomas Joseph. Exploiting virtual synchrony in distributed systems. In *Proc. of the 11th ACM Symp. on Operating Systems Principles (SOSP'87)*, 1987.

6 C. Cachin, R. Guerraoui, and L. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2nd Edition)*. Springer-Verlag, 2011.

7 Tushar Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

8 Partha Dutta, Rachid Guerraoui, Ron R. Levy, and Marko Vukolić. Fast access to distributed atomic memory. *SIAM Journal on Computing*, 39(8):3752–3783, 2010.

9 Cyntia Dwork, Nancy A. Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of ACM*, 35(2):288–322, 1988.

10 Rui Fan and Nancy Lynch. Efficient replication of large data objects. In *Proc. of the 17th Int. Symp. on Distributed Computing (DISC'03)*, 2003.

11 Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

12 Eli Gafni and Dahlia Malkhi. Elastic configuration maintenance via a parsimonious speculating snapshot solution. In *Proc. of the 29th Int. Symp. on Distributed Computing (DISC'15)*, 2015.

13 David Gifford. Weighted voting for replicated data. In *Proc. of the 7th ACM Symp. on Operating Systems Principles (SOSP'79)*, 1979.

14 Seth Gilbert, Nancy Lynch, and Alex Shvartsman. Rambo: A robust, reconfigurable atomic memory service for dynamic networks. *Distributed Computing*, 23(4), 2010.

15 Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programing Languages and Systems*, 13(1), 1991.

16 Leander Jehl and Hein Meling. The case for reconfiguration without consensus. In *Proc. of the 20th Int. Conf. on Principles of Distributed Systems (OPODIS'16)*, 2016.

17 Leander Jehl, Roman Vitenberg, and Hein Meling. Smartmerge: A new approach to reconfiguration for atomic storage. In *Proc. of the 29th Int. Symp. on Distributed Computing (DISC'15)*, 2015.

18 Leslie Lamport. On interprocess communication (part II). *Distributed Computing*, 1(1):203–213, 1986.

19 Leslie Lamport. The part-time parliament. *ACM Transactions Computer Systems*, 16(2):133–169, 1998.

20 Jean-Philippe Martin and Lorenzo Alvisi. A framework for dynamic Byzantine storage. In *Proc. of the 34th Int. Conf. on Dependable Systems and Networks (DSN'04)*, 2004.

21 Alexander Spiegelman, Idit Keidar, and Dahlia Malkhi. Dynamic Reconfiguration: A Tutorial. In *Proc. of the 19th Int. Conf. on Principles of Distributed Systems (OPODIS'15)*, 2015.

22 Brian White et. al. An integrated experimental environment for distributed systems and networks. In *Proc. of the 5th Symp. on Operating Systems Design and Implementations (OSDI'02)*, 2002.