Efficient Consensus-Free Weight Reassignment for Atomic Storage
Hasan Heydari, Guthemberg Silvestre, Luciana Arantes

To cite this version:
Hasan Heydari, Guthemberg Silvestre, Luciana Arantes. Efficient Consensus-Free Weight Reassignment for Atomic Storage. NCA 2021 - 20th IEEE International Symposium on Network Computing and Applications, Nov 2021, Virtual, France. hal-03454633

HAL Id: hal-03454633
https://enac.hal.science/hal-03454633
Submitted on 29 Nov 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Efficient Consensus-Free Weight Reassignment for Atomic Storage

Hasan Heydari  
*ENAC, Université du Toulouse*  
Toulouse, France  
heydari@enac.fr

Guthemberg Silvestre  
*ENAC, Université du Toulouse*  
Toulouse, France  
silvestre@enac.fr

Luciana Arantes  
*Sorbonne University, CNRS, Inria*  
Paris, France  
luciana.arantes@lip6.fr

Abstract—Weighted voting is a conventional approach to improving the performance of replicated systems based on commonly-used majority quorum systems in heterogeneous environments. In long-lived systems, a weight reassignment protocol is required to reallocate weights over time in order to accommodate performance variations accordingly. The weight reassignment protocol should be consensus-free in asynchronous failure-prone systems because of the impossibility of solving consensus in such systems. This paper presents an efficient consensus-free weight reassignment protocol for atomic storage systems in heterogeneous, dynamic, and asynchronous message-passing systems. An experimental evaluation shows that the proposed protocol improves the performance of atomic read/write storage implemented by majority quorum systems compared with previous solutions.

Index Terms—weighted voting, majority quorum system, replication, heterogeneous environment, dynamic distributed system

I Introduction

The atomic read/write storage (or simply atomic storage, a.k.a. atomic register [11]) is a fundamental building block for practical distributed storage and file systems (e.g., [2], [3]). Atomic storage allows concurrent processes, each possibly running a different algorithm, to share data atomically through a variable accessed by read/write (r/w) operations. Quorum systems [4] are a well-known abstraction for implementing atomic storage [5]. A quorum system is a collection of sets called quorums such that each one is a subset of processes, and the intersection property that states every two quorums always intersect should be satisfied. By implementing atomic storage using quorum systems, atomicity can be guaranteed using the intersection property [6]. Moreover, it is not required to execute r/w operations in all processes; each r/w operation should be executed by all processes of one quorum, improving the system’s fault tolerance and availability.

There exist many types of quorum systems such as grids [7], [8], trees [9], hierarchical [10], and the simple majority quorum system (SMQS) [11]. In the SMQS, every quorum consists of a strict majority of processes. Most atomic storages based on quorum systems (e.g., [5], [6], [12]) utilize the SMQS due to its simplicity and optimal fault tolerance; however, the SMQS can impact both quorum latency\(^1\) and throughput [11]. The reason for this performance impact is that an SMQS does not consider the heterogeneity of processes or network connections. If it takes such heterogeneity into account, its latency and throughput are likely to be improved.

Contrarily to SMQS, the weighted majority quorum system (WMQS) was proposed to cope with heterogeneity. In WMQS, each process is assigned a weight that is in accordance with the process’s latency or throughput determined by a monitoring system [14], [15]; every quorum consists of a set of processes such that the sum of their weights is greater than half of the total weight of processes in the system. The following example helps to grasp the difference between SMQS and WMQS in systems with heterogeneous latencies and throughput.

**Example 1.** Let \(p_1, p_2, p_3,\) and \(p_4\) be the processes comprising the system and \(c\) be a client. Consider the two following scenarios. For the first scenario, assume that the average round-trip latencies between the client and processes \(p_1, p_2, p_3,\) and \(p_4\) are 20\, ms, 45\, ms, 100\, ms, and 140\, ms, respectively. In another scenario, assume that the throughput of processes \(p_1, p_2, p_3,\) and \(p_4\) are 1000, 800, 400, and 200 operation/sec, respectively. Let \(1.4, 1.1, 0.9, 0.6\) be the assigned weights by the monitoring system to processes \(p_1, p_2, p_3,\) and \(p_4\), respectively. The quorum latency using SMQS is 100\, ms while using WMQS is 45\, ms (Figure 1). The throughput of the system based SMQS is 600 and 800 operation/sec, respectively\(^2\). Both scenarios show the advantage of using the WMQS over SMQS.

Although the WMQS improves the quorum latency in contrast to the SMQS, it has a significant drawback for real, dynamic, and long-lived systems, where the latencies and throughput of processes might change over time. Indeed, using time-invariant weights is not suitable for such systems, so the processes’ weights must be reassigned over time. However, reassigning processes’ weights is a challenging problem in

---

1Quorum latency for a request in a quorum system is the time interval between sending the request (to a quorum, some quorums, or a subset of processes) until receiving the responses from a quorum of processes [11], [13].

2Throughput is computed using *quoracle* library [11]; the code written using *quoracle* to compute throughput can be found in the full version of the paper [16].
dynamic asynchronous failure-prone systems due to the following requirements.

a) **Guaranteeing the atomicity property.** Weight reassignments and r/w operations might be concurrent. If it is the case, some r/w operations might be performed based on the most up-to-date weights, and others might be performed based on previous weights, not ensuring thus the atomicity property.

b) **Guaranteeing the liveness of atomic storage.** If one allows arbitrary weight reassignment, the system’s liveness might not be guaranteed. For instance, in Figure 1b, assume that process \( p_1 \)’s weight is reassigned to 2.7 while the weights of other processes are not reassigned. If process \( p_1 \) fails, no quorum can be constituted, leading to the loss of the system’s liveness. Note that the other processes’ weights cannot be reassigned anymore due to the asynchrony of the system. Indeed, process \( p_1 \) can be slow, and by reassigning other processes’ weights, two disjoint weighted quorums might be constituted.

c) **Demanding a consensus-free and wait-free solution.** For reassigning weights, a consensus-based protocol or similar primitives cannot be used because it is known that consensus is not solvable in asynchronous failure-prone systems [17]. Besides, the ABD protocol [6] showed that atomic storage could be implemented in static asynchronous systems in a wait-free [18] manner and without requiring consensus.

d) **Efficiency.** To have an efficient storage system, it is required to separate weight reassignment protocol from r/w protocols [19].

Atomic storage with consensus-based weight reassignment protocols (e.g., [20], [21]) does not satisfy the third requirement. Likewise, if consensus-based reconfiguration protocols (like RAMBO [12]) are adapted to be used as a weight reassignment protocol, the third requirement is not satisfied. Consensus-free reconfiguration protocols, like DynaStore [22], SpSn [23], and FreeStore [5] (more protocols can be found in [24], [25], [19]), proposed to change the set of processes that compromise the system by using two special functions: \( \text{join} \) and \( \text{leave} \). Servers can join/leave the system by calling these functions. Such protocols can be adapted to be used as a consensus-free solution for reassigning the processes’ weights. To do so, one can change \( \text{join} \) and \( \text{leave} \) functions to \( \text{increase} \) and \( \text{decrease} \) functions, respectively such that each server can request to increase/decrease its weight using \( \text{increase/decrease} \) functions. However, such protocols might create unacceptable states in which the liveness of atomic storage cannot be guaranteed (see Example 2).

**Example 2.** Let \( p_1, p_2, p_3, \) and \( p_4 \) be the processes comprising the system; also, let the initial weight of each process be one (Figure 2). Two concurrent requests \( \text{increase}(p_1, 1.4) \) and \( \text{decrease}(p_4, 0.7) \) are issued by processes \( p_1 \) and \( p_4 \), respectively, to increase \( p_1 \)’s weight by 1.4 and to decrease \( p_4 \)’s weight by 0.7. Each request creates an intermediate (auxiliary) state. Although each of created intermediate states is acceptable, their combination is unacceptable because the system might not be live (consider the case when process \( p_1 \) fails).

**Fig. 2:** An example to show that reconfiguration protocols (like DynaStore, SpSn, and FreeStore) might create unacceptable states if they are adapted to be used as a weight reassignment protocol.

SmartMerge [26] is the only consensus-free reconfiguration protocol that avoids creating unacceptable states; however, for each r/w operation, it requires communicating with a quorum of processes to find the most up-to-date configuration. Therefore, it might incur significant performance losses in terms of latency and throughput, i.e., it does not satisfy the fourth requirement since the r/w operations are not completely separated from the reconfiguration protocol. If SmartMerge is adapted to be used as a weight reassignment protocol, we still have the same problem.

To the best of our knowledge, no protocol is presented explicitly to solve weight reassignment in a consensus-free manner for atomic storage. This paper presents a novel and efficient consensus-free weight reassignment protocol that autonomously reassigns the processes’ weights for atomic storage (the atomic storage is based on the ABD protocol). The weight reassignment protocol avoids creating unacceptable states. In contrast to other solutions, the distinguishing feature of our protocol is that for executing each r/w operation, it is not required to communicate with a quorum of processes to find the most up-to-date processes’ weights leading to efficiency improvement. To evaluate the performance of an atomic storage based on our weight reassignment protocol, we compared our approach to atomic storages based on (1) the (static) ABD that uses an SMQS, (2) RAMBO, and (3) SmartMerge. Our experimental results show that our approach is 38%, 17%, 27% more efficient than the (static) ABD, RAMBO, and SmartMerge, respectively.

**Organization of the paper.** Section II presents the system model and some preliminary definitions and properties used in the paper. In Section III, we describe our weight reassignment protocol. We present the dynamic atomic storage that utilizes our weight reassignment protocol in Section IV. The
performance evaluation is shown in Section V. We present the conclusion and future work in Section VI.

II Preliminaries

In this section, we present the system model of our paper. Also, we present the preliminary definitions and properties of our weight reassignment protocol and dynamic atomic storage system.

System Model

We consider a distributed system composed by two non-overlapping sets of processes— a finite set of $n$ servers $S = \{s_1, s_2, \ldots, s_n\}$ and an infinite set of clients $C = \{c_1, c_2, \ldots\}$. Each process has a unique identifier. Every client or server knows the set of servers. Clients access the storage system provided by servers by executing $r/w$ operations. The processes communicate by message passing, and the links reliably connect all pairs of processes. Processes are prone to crash failures. A process is called correct if it is not crashed. The system is asynchronous, i.e., we make no assumptions about processing times or message transmission delays. However, each process has access to a local clock; processes’ local clocks are not synchronized and do not have any bounds on their drifts, being nothing more than counters that keep increasing. The interactions between processes are assumed to take place over a timespan $\mathcal{T} \subset \mathbb{R}^+$ called the lifetime of the system.

Weight Reassignment Definitions and Properties

Views. During the system’s lifetime, a sequence of views $\sigma = (v_0, v_1, \ldots)$ is installed in the system to reassign servers’ weights. The system starts in view $v_0$ called the initial view. The successor (resp. predecessor) of any view $v_k$ for $0 \leq k$ (resp. $1 \leq k$) is $v_{k+1} = v_k.succ$ (resp. $v_{k-1} = v_k.pred$). We say a view $v$ is installed in the system if a few correct servers consider $v$ as their current view (see definition below). We denote the current view of any server $s_i$ by $s_i.cview$. Note that the current views of servers might be different from the installed view in the system. When a non-initial view $v_{k+1}$ ($0 \leq k$) is installed, we say that $v_{k+1}.pred$ was uninstalled from the system. At any time $t \in \mathcal{T}$, we define lastview to be the last view installed in the system. Since lastview is the last installed view in the system, lastview.succ = ⊥.

The weights of servers are not reassigned during any view $v_k$ ($0 \leq k$) and might be reassigned at the time of uninstalling $v_k$ and installing $v_{k+1}$.

Installing a view. To install a view in the system, at least one server should request it. Each server $s_i$ can only request to install view $s_i.cview.succ$. To do so, $s_i$ sends a message $(\text{change_view}, s_i.cview.succ)$ to other servers. Each server $s_j$ sends each received message $(\text{change_view}, v)$ to other servers if it had not sent such a message previously. Then, each server $s_i$ sends message $(\text{state_update}, s_i.cview, w)$ to other servers, where the last parameter stands for $s_i$’s weight in $s_i.cview$ (we explain the algorithm for changing the views in further detail in Section III). Each server $s_i$ can install view $s_i.cview.succ$ as soon as receiving messages $(\text{state_update}, *, *, *, v, w)$ from a weighted majority of servers with views $v = s_i.cview$. As soon as at least one server installs a view $v$ such that $\forall s \in S : s.cview \leq v$, we say that view $v$ is installed in the system.

Comparing two views. We say that view $w$ is more up-to-date than view $v$ if the following recursive function returns yes by passing $(v, w)$ as input. We use the notation $v < w$ to state that view $w$ is more up-to-date than view $v$.

\begin{verbatim}
function more_up_to_date(v, w)
    v ← v.succ
    if $v = w$ then return yes
    else if $v = \bot$ then return no
    else return more_up_to_date(v, w)
\end{verbatim}

Our protocol’s assumptions and properties. The following assumptions and properties are required in our weight reassignment protocol. From now on, $\omega_l$, $\omega_u$, $\omega_T$, and $f$ state the lower bound of servers’ weights, the upper bound of servers’ weights, the total weight of servers, and the maximum number of failed servers, respectively.

Assumption 1 The initial weight of each server is equal to one. Formally, $\forall s_i \in S : s_i.v.weight = 1$.

Assumption 2 The values of $\omega_l$ and $\omega_u$ are $n/(2 \times (n - f))$ and $n/(2f)$, respectively.

Assumption 3 In each quorum, the total weight of servers is greater than $n/2$.

Assumption 4 The number of views that are requested to be installed in the system is finite. Formally, $|\sigma| = m$, where $m \in \mathbb{N}$ and $\sigma$ is the sequence of installed views.

Assumption 4 is used in all reconfiguration protocols presented for asynchronous systems such as RAMBO, DynaStore, SpSn, FreeStore, and SmartMerge. Its reason is that it is impossible to reconfigure a storage system infinitely many times while guaranteeing the liveness of the storage system [27].

Property 1 The total weight of servers is bounded by $n$ in any view $v \in \sigma$. Formally, $\forall v \in \sigma : \sum_{s \in S} s.v.weight \leq n$.

Property 2 The weight of each server in every view should be greater than $\omega_l$ and less than $\omega_u$. Formally, $\forall v \in \sigma, \forall s_i \in S : \omega_l < s_i.v.weight < \omega_u$.

If a system relies on Assumption 2, Assumption 3, Property 1, and Property 2, we can show that there is a quorum of correct servers, even if $f$ servers crash (in the worst case, $f$ servers crash so there are $n - f$ correct servers, each one with weight $\omega_l$; since $(n - f) \times \omega_l > n/2$, at least a quorum of servers can be constituted). Consequently, unacceptable states (like in Example 2) are not created.
Assumptions related to WMQS. The following assumptions are required to have performance gains by using WMQS.

Assumption 5 During the system’s lifetime: \(2f + 1 \leq n\).

This assumption is the same as the one used in other weight reassignment protocols, like WHEAT [28] and AWARE [14]. Indeed, this assumption states that there are a few additional spare servers, enabling the system to make progress without needing to access a majority of servers.

Assumption 6 Constant \(w_u\) should be defined in such a way that \(1 \leq w_u\).

The goal of WMQS is to constitute quorums with a minority of high-weighted servers to improve performance. To constitute a quorum with a majority of servers, it is required that \(1 \leq w_u\). The reason for having such a requirement is as follows. Due to Assumption 3, the total weight of servers is greater than \(n/2\) in each quorum. To have a quorum with a minority of servers, it is necessary (but not sufficient) to have at least one server with a weight greater than equal to 1.

Monitoring system. In order to reassign servers’ weights, the latencies of the server to server and client to server communications should be monitored. To this end, each server uses a local monitor module (like the one presented in AWARE [14]) that is responsible for evaluating and gathering information about latencies and giving scores to servers. We denote the latency score of server \(s_k\) computed using the monitoring system of server \(s_i\) by \(s_i.lscores.s_k\) \((1 \leq k \leq n)\). Note that it might be possible that \(s_i.lscores.s_k \neq s_j.lscores.s_k\), at any time \(t\). Any server \(s_i\) can compare its latency with another server \(s_j\) using latency scores. For instance, from server \(s_i\)’s point of view, the latency of server \(s_j\) is greater than \(s_i\)’s latency if \(s_i.lscores.s_i < s_i.lscores.s_j\).

Dynamic Storage Definitions and Properties

Views vs. r/w operations. At any time \(t \in T\), r/w operations can only be executed in view lastview. At the time of uninstalling any view \(v\), r/w operations are disabled on servers with view \(v\). The operations are enabled after installing view \(v.succ\).

Definition (Atomic register [29]). Assume two read operations \(r_1\) and \(r_2\) executed by correct clients. Consider that \(r_1\) terminates before \(r_2\) initiates. If \(r_1\) reads a value \(\alpha\) from register \(R\), then either \(r_2\) reads \(\alpha\) or \(r_2\) reads a more up-to-date value than \(\alpha\).

Dynamic storage. A dynamic storage satisfies the following properties: (1) the r/w protocols should implement an atomic register (Definition II), (2) every r/w operation executed by a correct client eventually terminates, (3) the r/w operations that are disabled on servers to install a view will eventually be enabled, (4) if any server \(s_i\) installs a non-initial view \(v\), some server has requested to install view \(v\), (5) reassigning weights are possible during the lifetime of the system, i.e., the weight reassignment protocol satisfies the liveness property.

III Weight Reassignment Protocol

In this section, we describe our weight reassignment protocol. The protocol has two essential dependent algorithms: pairwise weight reassignment and view changer. In the following, we describe these algorithms starting with pairwise weight reassignment. Then, we present the main properties of the weight reassignment protocol.

Pairwise Weight Reassignment

The default weight of servers in each view is one. However, for each succeeding view, any server might reassign its default weight. As a result, each server might have different weights in distinct views. Each server \(s_i\) to reassign its default weight associated with a view \(v\) \((s_i.cview < v)\) should participate in at least one pairwise weight reassignment. Pairwise weight reassignment is an algorithm in which two servers collaborate to reassign their weights associated with a view. Each pairwise weight reassignment \(pwr\) is characterized by a quadruple \((pwr.receiver, pwr.sender, w, v)\) such that for view \(v\), a server called \(pwr.sender\) decreases its weight associated with view \(v\) by weight \(w\) and sends \(w\) in a message to another server called \(pwr.receiver\) that has lower latency; \(pwr.receiver\) increases its weight associated with \(v\) by weight \(w\) after receiving the message containing \(w\). In this way, the servers that have lower latencies might become high-weighted servers leading to improving the performance.

Each server should satisfy the lower and upper bounds defined in Assumption 2 and Property 2. In other words, a server does not participate in a pairwise weight reassignment if its weight does not meet the lower and upper bounds. Besides, the total weight of the system is not changed by reassigning the weights in a pairwise manner; therefore, Property 1 can be satisfied as well. Consequently, unacceptable states cannot be created.

The pseudo-code of pairwise weight reassignment can be found in Algorithm 1. Each pairwise weight reassignment is started by sending a request issued by a server that wants to be the \(pwr.receiver\) (Lines 5-12). For simplicity, we assume that every requested weight is equal to a constant \(\epsilon\), i.e., for every pairwise weight reassignment \(pwr = (s_i, *, *, w, *)\), \(w = \epsilon\). Moreover, we assume that each server \(s_i\) can participate in a pairwise weight reassignment \(pwr\) as the \(pwr.receiver\) if \(v = s_i.cview.succ\), where \(pwr = (s_i, *, *, v)\). In other words, each server can only be a \(pwr.receiver\) for its succeeding view. Server \(s_i\) should meet the following conditions to be allowed to send a request to server \(s_j\) to start a pairwise weight reassignment for view \(v\).

C1R) View \(v\) should be equal to the succeeding view of server \(s_i\). Formally, \(s_i.cview.succ = v\) (Lines 6 and 12).
C2R) Server \(s_i\) has not participated in any operation related to \(s_i.cview.succ\). This can be ensured by a variable of Algorithm 2 (Lines 7-8).
C3R) Each server is allowed to send a request to another server that has a greater latency score. Formally, \(s_i.lscores.s_i < s_i.lscores.s_j\) (Line 9).
C4R) Each server is allowed to send a request if its weight does not exceed the upper bound \( w_u \) defined in Property 2. Formally, \( \text{get\_weight}(\text{cview.succ}) + \text{get\_requested\_weight}(\text{cview.succ}) + \epsilon < w_u \). Function \( \text{get\_weight}(v) \) is used to determine the weight associated with view \( v \), and \( \text{get\_requested\_weight}(v) \) is a function to determine the total weight of requested weights that their responses have not received yet for view \( v \) (Line 10).

If the above conditions are met, server \( s_i \) is allowed to send a request by a message \( \langle \text{propose\_pwr}, s_i, \text{cview.succ}, \epsilon \rangle \) to server \( s_j \) (Line 12). By sending this message, we say that server \( s_i \) proposes a pairwise weight reassignment to server \( s_j \) for view \( v = s_i, \text{cview.succ} \). Besides, server \( s_i \) adds the pairwise weight reassignment to a set \( s_i,pwr\_requests \) to meet C4R (Line 11). Each server \( s_i \) meets the following conditions by receiving any message \( \langle \text{propose\_pwr}, v, \epsilon \rangle \) from server \( s_j \).

C1S) View \( v \) should be as up-to-date as \( s_i, \text{cview.succ} \). Formally, \( s_i, \text{cview.succ} \leq v \) (Line 15).

C2S) Server \( s_i \) has not participated in any operation related to \( s_i, v \) (Lines 16-17).

C3S) The latency score of \( s_i \) should be greater than \( s_j \). Formally, \( s_i, \text{lscores}, s_j < s_i, \text{lscores}, s_i \) (Line 18).

C4S) Each server is allowed to accept a request if its weight does not get less than \( w_l \) to satisfy Property 2. Formally, \( w_l < \text{get\_weight}(\text{view.succ}) - \epsilon \) (Line 19).

If the above conditions are met, server \( s_i \) executes a command \( pwr$s \leftarrow pwr$s \cup \{ (s_j, s_i, v, -\epsilon) \} \) to store the pairwise weight reassignment associated with view \( v \) (Line 20). Then, server \( s_i \) sends message \( \langle \text{accept\_pwr}, v, \epsilon \rangle \) to server \( s_j \) (Line 21). By sending this message, we say that server \( s_i \) accepts the pairwise weight reassignment. Moreover, the pairwise weight assignment terminates for server \( s_i \). For simplicity, we omitted the part that server \( s_i \) does not accept the pairwise weight reassignment.

Server \( s_i \) meets conditions C1R and C2R by receiving any message \( \langle \text{accept\_pwr}, v, \epsilon \rangle \) from server \( s_j \) (Lines 23-25). If the conditions are met, server \( s_i \) executes a command \( pwr$s \leftarrow pwr$s \cup \{ (s_i, s_j, v, \epsilon) \} \) to store the pairwise weight reassignment associated with view \( v \) (Line 26). Also, server \( s_i \) removes the terminated pairwise weight reassignment from set \( pwr\_requests \) (Line 27). At this point, we say that the pairwise weight reassignment terminates for server \( s_i \).

**View Changer**

Each server can request to change the installed view in the system and change its current view by using an algorithm called the view changer algorithm. Algorithm 2 is the pseudocode of the view changer algorithm. In the following, we describe how this algorithm works.

**How servers can request to change a view.** Each server \( s_i \) for each view \( v \) has a timeout (Line 1 of Algorithm 2). When the timeout of view \( s_i, \text{cview} \) finishes, server \( s_i \) sends a request to other servers to change \( s_i, \text{cview} \) to \( s_i, \text{cview.succ} \) and stores \( s_i, \text{cview.succ} \) in a set denoted by \( s_i,\text{rchange\_views} \) (Lines 12-13). Such a request is sent by a message \( \langle \text{change\_view}, s_i, \text{cview.succ} \rangle \). Note that in practice, such a timeout should be big enough so that the views are changed rarely to satisfy Assumption 4.

**How a server can change its current view.** Each server \( s_i \), by receiving any message \( \langle \text{change\_view}, v \rangle \), stores view in a set denoted by \( s_i,\text{rchange\_views} \) (Line 30). As soon as \( s_i, \text{cview.succ} \in s_i,\text{rchange\_views} \), server \( s_i \) starts to change its view (Line 15). To do so, server \( s_i \) must do the following steps: (S1) Sending message \( \langle \text{change\_view}, v \rangle \) to other servers if it had not been sent yet. Set \( \text{rchange\_views} \) is used to store sent messages tagged with \( \text{change\_view} \) to be sure that a message is not sent more than once (Line 18). (S2) Disabling r/w operations (Line 19). (S3) Informing Algorithm 1 that some operations related to view \( s_i, \text{cview.succ} \) are processing to safety Conditions C2R and C2S (Line 20). (S4) Updating the states (registers) of servers (Lines 21-26). Each server has a register; in this step, the registers of servers with view \( v \) are synchronized. To do so, server \( s_i \) reads its register state (say, \( \varsigma \)) (Line 21). Then, server \( s_i \) sends message \( \langle \text{state\_update}, \varsigma, s_i, \text{cview}, s_i, \text{cview.weight} \rangle \) to other servers (Line 22). Server \( s_i \) waits until receiving messages \( \langle \text{state\_update}, *, s_i, \text{cview}, * \rangle \) from a weighted majority of servers (Lines 24). Finally, server \( s_i \) computes and stores the new state of its register (Lines 25-27). (S5) Changing the view (Line 28). (S6) Enabling r/w operations (Line 29).

**Example 3.** Figure 3 illustrates an example of executing some pairwise weight reassignments and the view changer algorithm. In this example, \( S = \{ s_1, s_2, s_3, s_4, s_5 \} \). Server \( s_1 \) proposes a pairwise weight reassignment to server \( s_3 \) and another one to server \( s_5 \) for view \( v_2 \). Also, server \( s_2 \) proposes a pairwise weight reassignment to server \( s_5 \) for view \( v_2 \). The proposed pairwise weight reassignments are accepted by servers \( s_4 \) and \( s_5 \). At time \( t \), the timeout of view \( v_1 \) for server \( s_3 \) finishes. Then, server \( s_1 \) sends a message tagged with \( \text{change\_view} \) to other servers. After receiving server \( s_1 \)'s message, to change the view, other servers send message \( \text{change\_view} \) as well. Server \( s_3 \) is the first server that changes its view from \( v_1 \) to \( v_2 \) at time \( t \); after that, other servers change their views as well. The weight of servers \( s_1, s_4, \) and \( s_5 \) in view \( v_2 \) are \( 1 + 2\epsilon, 1 - \epsilon, \) and \( 1 - 2\epsilon \), respectively. Although the pairwise weight reassignment proposed by server \( s_2 \) is accepted, it does not affect \( s_2 \)'s weight because \( s_2 \) had participated in view \( v_2 \) at the time of receiving the accept message; accordingly, the total weight of servers in view \( v_2 \) is \( w_1 - \epsilon \) until view \( v_2 \) is uninstalled. Since server \( s_3 \) has not participated in any pairwise weight reassignment, its weight in view \( v_2 \) is equal to its default weight.

**Properties of the Weight Reassignment Protocol**

The most important property of the presented weight reassignment protocol is that each server knows its up-to-date weight. Therefore, if a client sends a r/w request to a server, the server can include its weight to its response. Then, the client can decide whether a quorum of servers is constituted based
Algorithm 1 Pairwise weight reassignment - server $s_i$

variables
1) $pwrs \rightarrow \emptyset$
2) $pwr_requests \rightarrow \emptyset$

functions
3) $\text{get\_weight(view)} \equiv 1 + \sum \{w \mid (s,*,v,w) \in pwrs \text{ and } view = v\}$
4) $\text{get\_requested\_weight(view)} \equiv \sum \{w \mid (s,*,v,w) \in pwr_requests \text{ and } view = v\}$

while forever
5) atomic
6) $cview \leftarrow \text{get\_cview()} \text{ of Algorithm 2}$
7) $\text{dirty\_views} \leftarrow \text{get\_dirty\_views()} \text{ of Algorithm 2}$
8) if $cview.succ \notin \text{dirty\_views}$
9) if $\exists s_j : s_i,lscores.s_i < s_i,lscores.s_j$
10) if $\text{get\_weight(cview.succ)} + \text{get\_requested\_weight(cview.succ)} + \epsilon > w_u$
11) $\text{pwr\_requests} \leftarrow \text{pwr\_requests} \cup \{(s_i,s_j,cview.succ,\epsilon)\}$
12) send message ($\text{propose\_pwr},cview.succ,\epsilon$) to server $s_j$

upon receipt of message ($\text{propose\_pwr},view,\epsilon$) from server $s_j$
14) atomic
15) $cview \leftarrow \text{get\_cview()} \text{ of Algorithm 2}$
16) $\text{dirty\_views} \leftarrow \text{get\_dirty\_views()} \text{ of Algorithm 2}$
17) if $\text{view} \notin \text{dirty\_views}$
18) if $s_i,lscores.s_i < s_i,lscores.s_j$
19) if $w_i < \text{get\_weight(view)} - \epsilon$
20) $pwrs \leftarrow pwrs \cup \{(s_i,s_j,view,\epsilon)\}$
21) send message ($\text{accept\_pwr},view,\epsilon$) to server $s_j$

upon receipt of message ($\text{accept\_pwr},view,\epsilon$) from server $s_j$
23) atomic
24) if $cview.succ = \text{view}$
25) if $\text{view} \notin \text{dirty\_views}$
26) $pwrs \leftarrow pwrs \cup \{(s_i,s_j,view,\epsilon)\}$
27) $\text{pwr\_requests} \leftarrow \text{pwr\_requests} \setminus \{(s_i,s_j,view,\epsilon)\}$

$\triangleright$ a set for storing pairwise weight reassignments
$\triangleright$ a set for storing requested pairwise weight reassignments
$\triangleright$ a function to compute $s_i,view,weight$
$\triangleright$ atomic execution of lines 6-12; $s_i$ tries to be a $\text{pwr\_receiver}$
$\triangleright$ atomic execution of lines 14-21
$\triangleright$ view should be as up-to-date as $cview.succ$ (C1S)
$\triangleright$ atomic execution of lines 23-27
$\triangleright$ meeting Condition C1R

$\triangleright$ $s_i$ is the $\text{pwr\_sender}$
$\triangleright$ $s_i$ should not participate in any operation related to $cview.succ$ (C2R)
$\triangleright$ $s_i$ should not participate in any operation related to $\text{view}$ (C2S)
$\triangleright$ $s_i$ should has a lower latency than $s_j$ (C3S)
$\triangleright$ $s_i$’s weight should not get less than $\omega_i$ (C4S)
$\triangleright$ $s_i$’s weight should not exceed $\omega_i$ (C5S)

$\triangleright$ removing the terminated $\text{pwr}$ form $\text{pwr\_requests}$

Fig. 3: An example of executing some pairwise weight reassignments and the view changer algorithm. The weight of each server in view $v_2$ is shown at the top of the view.

IV Read/Write Protocols

In this paper we extend the (static) ABD protocol [6] to present a dynamic weighted atomic storage system that provides atomic r/w protocols. Algorithm 3 describes r/w protocols executed by clients. Besides, Algorithm 4 describes how a server processes r/w operations. The main differences between the original ABD protocol and the extended version are as follows.

1) Each client has a variable $cview$ denoting its current view and initialized with view $v_0$ (Line 1 of Algorithm 3). Each client adds its current view to every r/w request; moreover,


Algorithm 2 View changer - server $s_i$

variables
1) $cview \leftarrow v_0$, set a timeout for $cview$  
2) $rchange_views \leftarrow \emptyset$  
3) $schange_views \leftarrow \emptyset$  
4) $state_updates \leftarrow \emptyset$  
5) $dirty_views \leftarrow \{v_0\}$  

functions
6) $\text{sum_weights}(view) \equiv \sum\{w \mid (s,\ast,\ast,\ast,\ast,\ast,\ast,\ast,\ast) \in state\_updates \text{ and } view = v\}$  
7) $\text{maxz}(view) \equiv \max\{(t \mid (s,\ast,\ast,\ast,\ast,\ast,\ast,\ast,\ast) \in state\_updates \text{ and } view = v\}$  
8) $\text{maxc}(view, ts) \equiv \max\{(c \mid (s,\ast,\ast,\ast,\ast,\ast,\ast,\ast,\ast) \in state\_updates \text{ and } view = v \text{ and } ts = t\}$  
9) $val, maxz, maxc(\text{view}, ts, cid) \equiv \{val \mid (s,\ast,\ast,\ast,\ast,\ast,\ast,\ast,\ast) \in state\_updates \text{ and } view = v \text{ and } ts = t \text{ and } cid = c\}$
10) $\text{get_cview()} \equiv cview$  
11) $\text{get_dirty_views()} \equiv \text{dirty_views}$  
12) if $\text{the \ timeout \ for \ view \ } s_i.cview \ \text{ finishes}$  
13) $\text{send message } (\text{change_view, cview.succ}) \text{ to other servers}$  
14) $\text{change_views} \leftarrow \text{change_views} \cup \{\text{cview.succ}\}$  
15) if $\text{cview.succ} \in \text{change_views}$  
16) $\text{send message } (\text{change_view, cview.succ}) \text{ to other servers}$  
17) $\text{disable the execution of } r/w \text{ operations of Algorithm 4}$  
18) $\text{dirty_views} \leftarrow \text{dirty_views} \cup \{\text{cview.succ}\}$  
19) $\text{wait until } n/2 < \text{sum_weights}(\text{cview})$  
20) $\text{maxz} \leftarrow \text{max_timestamp}(\text{cview})$, $\text{maxc} \leftarrow \text{max_c}(\text{cview}, \text{maxz})$  
21) $\text{set ts, cid, val} (\text{maxz}, \text{maxc}, \text{val})$ of Algorithm 4  
22) $\text{send message } (\text{state_update, (val, ts, cid, cview, cview.weight)})$ to other servers  
23) $\text{state_updates} \leftarrow \text{state_updates} \cup \{(s_i, \text{val, ts, cid, cview, cview.weight})\}$  
24) $\text{wait until } n/2 < \text{sum_weights}(\text{cview})$  
25) $\text{maxz} \leftarrow \text{max_timestamp}(\text{cview})$, $\text{maxc} \leftarrow \text{max_c}(\text{cview}, \text{maxz})$  
26) $\text{set ts, cid, val} (\text{maxz}, \text{maxc}, \text{val})$ of Algorithm 4  
27) $\text{cview} \leftarrow \text{cview.succ}$, set a timeout for $\text{cview}$  
28) $\text{enable the execution of } \text{r/w} \text{ operations}$  

upon receipt of message $(\text{change\_view, view})$
30) $\text{rchange_views} \leftarrow \text{rchange_views} \cup \{\text{view}\}$  

upon receipt of message $(\text{state\_update, (val, ts, cid, view, weight)})$ from server $s$
31) $\text{state_updates} \leftarrow \text{state_updates} \cup \{(s, \text{val, ts, cid, view, weight})\}$

Each client sends its r/w requests to all servers (Line 5 of Algorithm 3).

2) After receiving each r/w request $r$, each server $s$ determines its current view $(s,cview)$ by calling function $\text{get_cview()}$ from Algorithm 2, and sets a variable weight to $\perp$ (Lines 5 and 9 of Algorithm 4). For write requests, if the current view of each request is the same as $s,cview$, server $s$ executes the request (Lines 11-12 of Algorithm 4). Additionally, server $s$ resets the value of variable weight by calling function $\text{get_weight}(s,cview)$ from Algorithm 1 (Line 13 of Algorithm 4). Similarly, for read requests, if the current view of each request is the same as $s,cview$, server $s$ resets the value of variable weight (Line 7 of Algorithm 4). Then, server $s$ adds $s,cview$ and weight to its response of the client that issued the request.

3) Each client updates its current view as soon as it receives a more up-to-date view than its current view and restarts the executing operation (Lines 12-14 and 30-32 of Algorithm 3).

4) Clients consider the weights of servers to decide whether a quorum is constituted (Lines 15 and 33 of Algorithm 3).

Correctness

We can prove our storage system satisfies the properties defined in Section II (see [16] for the proofs). In other words, every r/w operation executed by a correct client eventually terminates, and the r/w protocols of our weighted storage implement an atomic r/w register (Definition II).

V Performance Evaluation

In this section, we present a performance evaluation of the atomic storage based on our weight reassignment protocol to quantify its quorum latency when compared with atomic storage systems based on the following cases: (1) the (static) ABD [6] that uses an SMQS, (2) RAMBO [12] (a consensus-based reconfiguration protocol), and (3) SmartMerge [5] (a consensus-free reconfiguration protocol). We selected SmartMerge because it avoids unacceptable states in contrast to other consensus-free reconfiguration protocols (e.g., [5], [22], [23]). We implemented prototypes of the ABD, RAMBO, and SmartMerge protocols in the python programming language. Besides, we used KOLPASS [30], a fully distributed network emulator, to create the network and links’ latencies.
As we explained in Section I, reconﬁguration protocols present two special functions: join and leave. Servers can join/leave the system by calling these functions. Reconﬁguration protocols require to be adapted to be used as weight requirement protocols. To do so, we change join and leave functions of RAMBO and SmartMerge to increase and decrease functions, respectively. Each server can request to increase/decrease its weight using increase/decrease functions. Particularly, each server can call increase and decrease functions every δ unit of time (0 < δ is a constant) as follows to increase/decrease its weight. Assume that the latency score of server s are $l_S$ and $l_S'$ respectively at time $t$ and $t' = t + δ$. Also, assume that the total latency scores of servers computed by $s$ are $L_S$ and $L_S'$ respectively at time $t$ and $t'$. Server $s$ calls function increase (resp. decrease) to increase (resp. decrease) its weight at time $t'$ if $l_S'/L_S + \tau < l_S'/L_S$ (resp. $l_S'/L_S + \tau < l_S'/L_S$), where $\tau$ is a threshold for changing weights.

We used one 1.8 GHz 64-bit Intel Core i7-8550U, 32GB of RAM machine. KOLLAPS executes each server and client in a separate Docker container [31], and the containers communicate through the Docker Swarm [32]. We set the numbers of servers and clients to ﬁve and ten, respectively. Moreover, at most, one server can fail ($f = 1$). Each client sends a new r/w request as soon as receiving the response of the previously sent r/w request. Since there is no difference between r/w protocols regarding the number of communication rounds in our r/w protocols, we set the r/w ratio to 0.5.

The duration of each run is 200 seconds. In each run, latencies of links are changed every $\Delta = 10$ seconds while the processes are unaware of the value of $\Delta$; we set $\epsilon = 0.1$. We executed 100 runs and computed the average of the results that is depicted in Figure 4. The average quorum latencies of the ABD, RAMBO, SmartMerge and our protocol are 139, 118, 128, and 101 milliseconds, respectively. The ABD protocol requires the responses of three processes to decide whether a quorum is constituted while other protocols might constitute their quorums by two processes. Therefore, the quorum latencies of other protocols are less than the ABD on average. In the RAMBO protocol, some views might be active at a time, while in our protocol, there is only one installed view at any time; hence, our protocol outperforms the RAMBO protocol on average. In the SmartMerge protocol, servers might pass intermediate views to install a new view; besides, for every r/w operation, it is required to communicate with a quorum of processes to be sure that the r/w operation is executed with the most up-to-date weights. However, servers with the view equal to lastview directly change their views to the new view in our protocol. Also, each server $s$ knows its weight, i.e., $s$ does not need to communicate with others to determine its most up-to-date weight. Hence, the quorum latency of our protocol is less than SmartMerge on average.

Algorithm 3 ABD - client $c_i$

variables
1. $opCnt \leftarrow 0$, $cview \leftarrow t_0$
functions to r/w the atomic storage
2. $read() \equiv read_write(.)$
3. $write(value) \equiv read_write(value)$

function $read_write(value)$

ABD Phase 1
4. $opCnt \leftarrow opCnt + 1$
5. send ($read, opCnt, cview$) to all servers
6. $msgs \leftarrow \emptyset$
7. $repeat$
8. upon receipt of message ($readack, val, ts, cid, opCnt, v, w$)
9. if $cview = v$
10. $msgs \leftarrow msgs \cup \{(val, ts, cid, opCnt, v, w)\}$
11. else
12. if $cview < v$
13. $cview \leftarrow v$
14. $read_write(value)$ ▶ restart the operation
15. until $n/2 \leq \sum \{|w| \,(v, *, *, *, *, w) \in msgs\}$
16. if $value = \bot$
17. $maxts \leftarrow \max \{|ts| \, (v, *, *, *, *, *) \in msgs\}$
18. $maxcid \leftarrow \max \{|cid| \, (v, *, *, *, *, *) \in msgs\}$
19. $value \leftarrow \{val \mid (val, ts, cid, *, *, *) \in msgs \wedge ts = maxts\}$
20. else
21. $maxts \leftarrow \max \{|ts| \, (v, *, *, *, *, *) \in msgs\} + 1$
22. $maxcid \leftarrow c_i$, $maxval \leftarrow val$

ABD Phase 2
23. send ($write, value, maxts, maxcid, opCnt, cview$) to all servers
24. $msgs \leftarrow \emptyset$
25. $repeat$
26. upon receipt of message ($writeack, opCnt, v, w$)
27. if $cview = v$
28. $msgs \leftarrow msgs \cup \{(opCnt, v, w)\}$
29. else
30. if $cview < v$
31. $cview \leftarrow v$
32. $read_write(value)$ ▶ restart the operation
33. until $n/2 \leq \sum \{|w| \,(v, *, *, w) \in msgs\}$
34. return $value$

Algorithm 4 ABD - server $s_i$

variables
1. $ts \leftarrow 0$, $cid \leftarrow 0$, $val \leftarrow \bot$
functions
2. $get_ts() \equiv ts$
3. $get_ts_cid_val() \equiv (ts, cid, val)$
4. $set_ts_cid_val(t, c, v) \equiv ts \leftarrow t; cid \leftarrow c; val \leftarrow v$
upon receipt of message ($read, cnt, v$) from client $c$
5. $cview \leftarrow get_cview$ from Algorithm 2, $weight \leftarrow \bot$
6. if $cview = v$
7. $weight \leftarrow get_weight(cview)$ from Algorithm 1
8. send ($readack, val, ts, cid, cview, weight$) to client $c$
upon receipt of message ($write, val', ts', cid', cnt, v$) from client $c$
9. $cview \leftarrow get_cview$ from Algorithm 2, $weight \leftarrow \bot$
10. if $cview = v$
11. if $ts' > ts$ or $(ts' = ts$ and $cid' > cid)$
12. $ts \leftarrow ts'$, $cid \leftarrow cid'$, $val \leftarrow val'$
13. $weight \leftarrow get_weight(cview)$ from Algorithm 1
14. send ($writeack, cnt, cview, weight$) to client $c$
VI Conclusion and Future Work

In this paper, we present a novel consensus-free and crash fault-tolerant weight reassignment protocol that can be used to improve the performance of atomic read/write storage systems. The distinguishing feature of our protocol compared to previous solutions is that for executing each r/w operation, it is not required to communicate with a quorum of processes for finding the most up-to-date processes’ weights, providing better efficiency. The evaluation results show that our protocol outperforms other solutions. We assume that the set of servers does not change over time; however, the protocol can be extended to consider that servers can leave and new servers can join the system as future work. Besides, every client sends each of its requests to all servers. Working on using strategies for selecting a subset of servers to send requests for improving the network congestion can be another direction for future improvement. Extending the failure model to Byzantine failures could be another direction for future work as well.

References