How Hard is Asynchronous Weight Reassignment? (Extended Version)

Hasan Heydari, Guthemberg Silvestre, Alysson Bessani

How Hard is Asynchronous Weight Reassignment?

(Extended Version)

Hasan Heydari,* Guthemberg Silvestre,† and Alysson Bessani*
*LASIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal
†ENAC, University of Toulouse, France
hheydari@ciencias.ulisboa.pt, silvestre@enac.fr, anbessani@ciencias.ulisboa.pt

***Abstract*—The performance of distributed storage systems deployed on wide-area networks can be improved using weighted (majority) quorum systems instead of their regular variants due to the heterogeneous performance of the nodes. A significant limitation of weighted majority quorum systems lies in their dependence on static weights, which are inappropriate for systems subject to the dynamic nature of networked environments. To overcome this limitation, such quorum systems require mechanisms for reassigning weights over time according to the performance variations. We study the problem of node weight reassignment in asynchronous systems with a static set of servers and static fault threshold. We prove that solving such a problem is as hard as solving consensus, i.e., it cannot be implemented in asynchronous failure-prone distributed systems. This result is somewhat counter-intuitive, given the recent results showing that two related problems – replica set reconfiguration and asset transfer – can be solved in asynchronous systems. Inspired by these problems, we present two versions of the problem that contain restrictions on the weights of servers and the way they are reassigned. We propose a protocol to implement one of the restricted problems in asynchronous systems. As a case study, we construct a dynamic-weighted atomic storage based on such a protocol. We also discuss the relationship between weight reassignment and asset transfer problems and compare our dynamic-weighted atomic storage with reconfigurable atomic storage.**

***Index Terms*—distributed storage, weighted replication, atomic storage, asset transfer, reconfiguration, consensus**

I. Introduction

In the era of cloud computing, cryptocurrencies, and the internet of everything, distributed storage systems are required more than ever due to their fault tolerance and high availability. Ensuring consistency of distributed storage systems is a fundamental challenging problem in distributed computing. One well-known solution for such a problem is utilizing quorum systems [1]. A quorum system is a collection of sets called *quorums* such that each quorum is a subset of servers, and every two quorums *intersect*. Although many types of quorum systems exist, such as grids [2] and trees [3], most practical distributed storage systems (e.g., [4]–[7]) utilize the regular majority quorum system (MQS) due to its simplicity and optimal fault tolerance.

In MQS, every quorum consists of a strict majority of servers. Although MQS is simple and optimally fault-tolerant, it might be subject to poor latency and low throughput due to practical considerations such as replica heterogeneity [8]. To take such considerations into account, one can use the weighted majority quorum system (WMQS), in which each server is assigned a weight (a.k.a. vote or voting power) in accordance with its access latency or request processing capacity (throughput), as determined by a monitoring system [9], [10], and the assigned weights are used to determine whether a subset of servers constitutes a (weighted) quorum.

A significant limitation of the WMQS is its reliance on static weights, which are inappropriate for dynamic systems, where servers' performance might change over time [10], [11]. To overcome such a limitation, WMQS can be integrated with weight reassignment protocols for changing server weights over time according to performance variations.

The main goal of this paper is to study weight reassignment in an asynchronous system with a static set of servers and static fault threshold, where an available weighted quorum is guaranteed to exist. To this end, as a first step, we formally define the *weight reassignment* problem by which weight reassignment requests can be issued and processed. We then prove that consensus can be reduced to the weight reassignment problem, i.e., a solution to the weight reassignment problem can be used to solve consensus. Consequently, *the weight reassignment problem cannot be implemented in asynchronous failure-prone systems*.

To cope with such an impossibility, we introduce a restricted version of weight reassignment called *pairwise weight reassignment*, in which the reassignments can only be done in a pairwise way. More precisely, in the pairwise weight reassignment, the total weight of servers remains constant, and a server gains a weight Δ if and only if another server loses Δ. Reassigning weights in such a way is similar to transferring assets in 1-asset transfer.[1] Somewhat surprisingly, although 1-asset transfer can be implemented in asynchronous failure-prone systems [12], we show that this is not the case for pairwise weight reassignment.

We further restrict the pairwise weight reassignment problem by mainly considering a restriction on the possible range of weights, naming it *restricted pairwise weight reassignment*. We show that such a restricted variant of the problem can

This is the extended version of a paper to appear at the 43rd IEEE International Conference on Distributed Computing Systems (ICDCS 2023).

[1] In the 1-asset transfer problem, there are some accounts, each of which is owned by a server; each server can transfer some of its assets to another server if its balance does not become negative.

be implemented in asynchronous failure-prone systems. As a case study, we construct a dynamic-weighted atomic storage incorporating a protocol solving this variant.

Our dynamic-weighted atomic storage is somewhat similar to reconfigurable atomic storage,[2] which can be implemented in asynchronous systems [13]–[17]. We further elaborate on similarities between these storage systems, discussing why the techniques used to implement reconfigurable atomic storage, e.g., generalized lattice agreement [18], cannot be used to implement dynamic-weighted atomic storage.

Contributions. The contributions of this paper are:

- We formalize the *weight reassignment problem* for systems with a static set of servers and fault threshold and prove it cannot be solved in asynchronous failure-prone systems.
- We introduce a restricted version of the problem called the *pairwise weight reassignment*, in which voting power is transferred between pairs of servers. Although similar to asset transfer, we show that this variant cannot be implemented in asynchronous failure-prone systems.
- We further restrict the problem to a *restricted pairwise weight reassignment* variant, which can be implemented in asynchronous failure-prone systems, and use it to build a dynamic-weighed atomic storage.
- We discuss the relationship between the *pairwise weight reassignment* with the asset transfer problem and compare our dynamic-weighted atomic storage with reconfigurable atomic storage.

Organization of the paper. Section II presents our system model and preliminary definitions. In Section III, we introduce the weight reassignment problem. The impossibility of implementing the weight reassignment problem in asynchronous failure-prone systems is presented in Section IV. Section V presents the restricted versions of the weight reassignment problem: the pairwise weight reassignment and the restricted pairwise weight reassignment. We also show that pairwise weight reassignment cannot be implemented in asynchronous failure-prone systems. Section VI implements the restricted pairwise weight reassignment, while in Section VII, we outline a dynamic-weighted atomic storage using this implementation. Sections VIII and IX present related work and concludes the paper, respectively.

II. Preliminaries

System model. We consider an asynchronous message-passing system composed of two non-overlapping sets of processes – a finite set of n servers $\mathcal{S}$ and an infinite set of clients Π. Every client or server knows the set of servers. At most f servers can crash, while any number of clients may crash. A process is called *correct* if it is not crashed. Each pair of processes is connected by a reliable communication link. Processes are sequential, i.e., a process never invokes a new operation before obtaining a response from a previous one. In the definitions and proofs, we make the standard assumption of the existence of a global clock not accessible to the processes.

[2]Reconfigurable atomic storage implements atomic storage in systems with the possibility of changing the set of servers over time.

Weighted majority quorums. Our work relies on weighted majority quorums [19]–[22], so we present their definition and one of their properties that plays an essential role throughout this paper.

Definition 1 (Weighted Majority Quorum System)**.** The weighted majority quorum system (WMQS) refers to a set of quorums where each quorum is composed of a set of servers whose total weight is greater than half of the total weight of all servers.

Since some minority of servers might have the majority of weights, proportionally smaller quorums can be constituted in WMQS in contrast to MQS, yielding to performance improvements. To guarantee the availability of a distributed system based on WMQS, a relationship between the servers' weights and f must be satisfied; otherwise, more than half of the voting power can be assigned to f or fewer servers [20]. The following property determines such a relationship.

Property 1 (Availability of WMQS)**.** A WMQS is available if the sum of the f greatest weights is less than half of the total weight of all servers.

Consensus. This paper shows the impossibility of solving two problems related to weight reassignment in asynchronous failure-prone systems. Our method of showing these results involves reducing consensus to each of these problems (comprehensive explanation of these reductions detailed in Sections IV and V). In consensus, each correct process proposes a value v through the invocation of `propose`(v), and the ultimate goal of processes is to decide upon a single value from the proposed values. The operation must return the decided value, and any algorithm that solves consensus must satisfy the following properties (e.g., [23]):

- Agreement. All correct processes must decide the same value.
- Validity. If all correct processes propose the same value v, they must decide v.
- Termination. All correct processes must eventually decide.

III. Weight Reassignment Problem

The *weight reassignment problem* aims to capture the safety and liveness properties that must be satisfied as servers' weights change over time. To formalize this problem, we first need to define the *change* data structure, which contains the essential information related to the outcome of a weight reassignment operation.

Change. Each process p_i (server or client) has a local counter denoted by lc_i. We define *change* as $\{\mathcal{S} \cup \Pi\} \times \mathbb{N} \times \mathcal{S} \times \mathbb{R}$, where the quadruple $\langle p_i, lc_i, s, \Delta \rangle$ indicates that the weight of server s is changed by Δ as an outcome of a reassignment request made by process p_i with local counter lc_i. For any

change $\langle *, *, s, \Delta \rangle$, by convention, "the weight of the change" refers to Δ, and we say that "the change is created for s".

The problem. We introduce a weight reassignment problem with the following operations:

- `reassign`(s, Δ), where s is a server, and Δ is a real number different from zero, and
- `read_changes`(s), where s is a server.

Each process can invoke `reassign`(s, Δ) to request changing the weight of server s by Δ. Without loss of generality, we assume that *only servers* invoke `reassign`. When an invocation of `reassign` is *completed* (see definition below), a change c corresponding to the invocation's outcome is created, and a message $\langle$`Complete`$, c\rangle$ is returned to the server that invoked the operation. Any process can invoke `read_changes` to learn about the set of changes created for a server, by which the weight of the server can be calculated. Each process must increment its local counter after each invocation of the `reassign` operation.

Definition 2 (*Completed* `reassign`)**.** Assume that a server s_i invokes `reassign`(s_j, Δ) when its local counter is lc_i. We say that the invocation is *completed* if there is a time after which the response of every invocation `read_changes`(s_j) contains a change $\langle s_i, lc_i, s_j, * \rangle$.

We define $\mathcal{C}_{s,t}$ as the set containing every change c created for server s such that the `reassign` operation led to the creation of c is completed at time t. It is straightforward to show that $\mathcal{C}_{s,t} \subseteq \mathcal{C}_{s,t'}$ for any server s and $t \leq t'$, and we say that $\mathcal{C}_{s,t'}$ is *more up-to-date* than $\mathcal{C}_{s,t}$ if $\mathcal{C}_{s,t} \subset \mathcal{C}_{s,t'}$. Further, for each server s, we assume there is a change defining the initial weight of s. Specifically, given w as the initial weight of s, we assume that `reassign`(s, w) is completed at time $t = 0$. We denote the weight of a server s at any time t by $\mathtt{W}_{s,t}$, where $\mathtt{W}_{s,t} \triangleq \sum_{\langle *,*,s,\Delta \rangle \in \mathcal{C}_{s,t}} \Delta$. We also denote the weight of a set of servers $A \subseteq \mathcal{S}$ by $\mathtt{W}_{A,t}$, where $\mathtt{W}_{A,t} \triangleq \sum_{s \in A} \mathtt{W}_{s,t}$. With these definitions, we are ready to define the weight reassignment problem.

Definition 3 (Weight Reassignment Problem)**.** Any algorithm that solves the weight reassignment problem must satisfy the following properties:

- Integrity. $\forall t \geq 0, \forall F \subset \mathcal{S}$ such that $|F| = f$, $\mathtt{W}_{F,t} < \frac{\mathtt{W}_{\mathcal{S},t}}{2}$.
- Validity-I. When the `reassign`(s, Δ) operation is completed, a change $\langle *, *, s, \Delta \rangle$ is created if Integrity is not violated; otherwise, a change $\langle *, *, s, 0 \rangle$ is created.
- Validity-II. If `read_changes`(s) is invoked at time t, a set containing $\mathcal{C}_{s,t}$ is returned as the response.
- Liveness. If a correct server s invokes `reassign` (resp. a correct process p invokes `read_changes`), the invocation will eventually be completed, and s (resp. p) will receive a message $\langle$`Complete`$, *\rangle$ (resp. a set of changes).

It is straightforward to see why the Liveness property is a part of the problem's definition. In the following, we discuss why the other properties are required.

- Integrity is a consequence of Property 1, which determines the relationship between the servers' weights and f, guaranteeing the system's availability over time.
- The second property states that a change must be created as the outcome of each `reassign` invocation. It also determines how the change must be created.
  Notice that Integrity might be violated if each invocation of `reassign`(s, Δ) is completed by creating a change $\langle *, *, s, \Delta \rangle$ (see example below). Hence, to avoid the violation of Integrity, an invocation `reassign`(s, Δ) might be aborted, i.e., a change $\langle *, *, s, 0 \rangle$ is created as the invocation's outcome.
- The third property determines what responses to a `read_changes` invocation are valid. Given any process that invokes `read_changes`(s) at any time $t \geq 0$, it is clear that a valid response must be as up-to-date as $\mathcal{C}_{s,t}$. On the other hand, due to asynchrony, it is impossible to guarantee that exactly $\mathcal{C}_{s,t}$ is returned as the response. Consequently, a valid response is one that contains $\mathcal{C}_{s,t}$.

Example 1. Let $\mathcal{S} = \{s_1, s_2, s_3, s_4\}$, $\Pi = \{c_1, c_2\}$, and $f = 1$. The following sets contain the initial weights: $\mathcal{C}_{s_1,0} = \{\langle s_1, 1, s_1, 1 \rangle\}$, $\mathcal{C}_{s_2,0} = \{\langle s_2, 1, s_2, 1 \rangle\}$, $\mathcal{C}_{s_3,0} = \{\langle s_3, 1, s_3, 1 \rangle\}$, and $\mathcal{C}_{s_4,0} = \{\langle s_4, 1, s_4, 1 \rangle\}$. Assume that server s_1 invokes `reassign`$(s_1, 1.5)$, which is completed at time t_1. Accordingly, a change $\langle s_1, 2, s_1, 1.5 \rangle$ is created as the outcome of the invocation. It is worth mentioning that this invocation cannot be completed by creating a change with zero weight, as Validity-I enforces the invocation to create a change with non-zero weight when there is no Integrity violation.

Client c_1 invokes `read_changes`(s_1) after t_1 and receives a set of changes $\mathcal{C}$ at time t_2, where $\mathcal{C} = \mathcal{C}_{s_1,0} \cup \{\langle s_1, 2, s_1, 1.5 \rangle\}$. Note that Validity-II is violated if c_1 receives $\mathcal{C}_{s_1,0}$ (or any other set than $\mathcal{C}$). Client c_1 can calculate the weight of s_1 using $\mathcal{C}$: the weight of s_1 equals 2.5. Server s_3 invokes `reassign`$(s_2, -0.5)$ after t_2, which is completed at time t_3. Notice that creating a change $\langle s_3, 2, s_2, -0.5 \rangle$ violates Integrity, so a change $\langle s_3, 2, s_2, 0 \rangle$ is created. If client c_2 invokes `read_changes`(s_2) after t_3, it will receive $\mathcal{C}' = \mathcal{C}_{s_2,0} \cup \{\langle s_3, 2, s_2, 0 \rangle\}$. It is important to note that servers are not allowed to invoke `reassign`$(*, 0)$ because the second parameter of `reassign` must be non-zero.

IV. Impossibility Result

We now show the weight reassignment problem introduced in the previous section cannot be implemented in asynchronous failure-prone systems. We start by presenting an insight into such an impossibility result.

Consider a system in which all correct servers invoke the `reassign` operation concurrently such that only one of the invocations can be completed by creating a change with non-zero weight. That is, creating two or more changes, each with non-zero weight, violates Integrity, meaning that it can make f servers have more than half of the total voting power in the system. Assume that invocations `reassign`$(s_1, \Delta_1), \ldots,$ `reassign`(s_n, Δ_n) create such a sit-

uation. One can take the following steps to solve consensus among servers:

1) each correct server s_i writes its proposal v_i to a single-writer multi-reader (SWMR) register $R[i]$ and invokes $\texttt{reassign}(s_i, \Delta_i)$, where $1 \leq i \leq n$, and
2) if a change with non-zero weight is created for s_j, the decided value is the one stored in $R[j]$.

Since the weight of only one of the created changes is non-zero, servers can decide the same value. Consequently, consensus can be reduced to the weight reassignment problem, which means that the weight reassignment problem cannot be implemented in asynchronous failure-prone systems [24].

Based on this insight, we design an algorithm presented in Algorithm 1, by which servers solve consensus using the weight reassignment problem, i.e., it reduces consensus to the weight reassignment problem. The algorithm is executed by each correct server s_i and provides a function – $\texttt{propose}(v_i)$ – by which s_i proposes a value v_i. We divide the servers into two disjoint sets, F and $\mathcal{S} \setminus F$, such that $F = \{s_1, s_2, \ldots, s_f\}$, and we assume that the initial weight of every server $s \in F$ (resp. $s \in \mathcal{S} \setminus F$) equals $\frac{n-1}{2f}$ (resp. $\frac{n+1}{2(n-f)}$). Notice that Integrity is satisfied with these initial weights. Further, there is a shared array of SWMR registers R of size n to store servers' proposals.

Each server s_i executes the $\texttt{propose}$ function. After storing its proposal in $R[i]$, s_i invokes $\texttt{reassign}(s_i, 0.5)$ (resp. $\texttt{reassign}(s_i, -0.5)$) if $s_i \in F$ (resp. $s_i \in \mathcal{S} \setminus F$). It is straightforward to see that two or more invocations of $\texttt{reassign}$ cannot be completed by creating changes with non-zero weights. For instance, if $\texttt{reassign}(s_1, 0.5)$ and $\texttt{reassign}(s_{f+1}, -0.5)$ are completed by creating changes $\langle s_1, 2, s_1, 0.5\rangle$ and $\langle s_{f+1}, 2, s_{f+1}, -0.5\rangle$ at time $t > 0$, then we have:

$$\texttt{W}_{F,t} = f \times \frac{n-1}{2f} + 0.5 = \frac{n}{2}$$

$$\frac{\texttt{W}_{\mathcal{S},t}}{2} = \frac{\texttt{W}_{F,t} + \texttt{W}_{\mathcal{S}\setminus F,t}}{2} = \frac{f \times \frac{n-1}{2f} + 0.5 + (n-f) \times \frac{n+1}{2(n-f)} - 0.5}{2} = \frac{n}{2},$$

which means that Integrity is violated.

In a loop, for each server $s_j \in \mathcal{S}$, s_i repeatedly invokes $\texttt{read\_changes}(s_j)$ to see the invocation of which server is completed by creating a change with non-zero weight. Because of Liveness, the loop will eventually terminate. Assume that the invocation of server s_j is completed by creating a change $\langle s_j, 2, s_j, \Delta\rangle$, where $\Delta \neq 0$. Consequently, s_i returns $R[j]$ as the decided value, and consensus among servers will be solved.

Theorem 1. Consensus can be reduced to the weight reassignment problem.

Corollary 1. The weight reassignment problem cannot be implemented in asynchronous failure-prone systems.

The proof of Theorem 1 and other theorems presented in the following sections can be found in the appendix.

Algorithm 1 Reducing consensus to the weight reassignment problem – server s_i.

▷ R is a shared array of SWMR registers with size n
▷ if $i \in \{1, 2, \ldots f\}$, $\texttt{W}_{s_i,0} = \frac{n-1}{2f}$; otherwise, $\texttt{W}_{s_i,0} = \frac{n+1}{2(n-f)}$
▷ s_i executes the $\texttt{propose}$ function

function $\texttt{propose}(v_i)$
1: $R[i] \leftarrow v_i$
2: **if** $i \in \{1, 2, \ldots, f\}$
3: $\quad\texttt{reassign}(s_i, 0.5)$
4: **else**
5: $\quad\texttt{reassign}(s_i, -0.5)$
6: $decided\_value \leftarrow \perp$
7: **repeat**
8: $\quad$**for** $j \in \{1, 2, \ldots n\}$
9: $\quad\quad\mathcal{C} \leftarrow \texttt{read\_changes}(s_j)$
10: $\quad\quad$**if** $\langle s_j, 2, s_j, \Delta\rangle \in \mathcal{C}$ such that $\Delta \neq 0$
11: $\quad\quad\quad decided\_value \leftarrow R[j]$
12: **until** $decided\_value \neq \perp$
13: **return** $decided\_value$

V. Restricting the Weight Reassignment Problem

The weight reassignment problem presented in Section III cannot be implemented in asynchronous failure-prone systems according to Corollary 1. In this section, we try to restrict that problem in order to find a variant that can be implemented in such a system model. We begin by keeping the total voting power constant by restricting the reassignments to be done in a pairwise manner. We call the resulting problem the *pairwise weight reassignment*.

A. Pairwise weight reassignment

Avoiding the Integrity violation is the main difficulty in solving the weight reassignment problem, so we focus on this property. Our first idea for restricting the problem is to ensure that the right-hand side of the inequality presented in the Integrity property and, consequently, the total weight of all servers, remains constant over time, i.e., at any time $t > 0$, $\texttt{W}_{\mathcal{S},t} = \texttt{W}_{\mathcal{S},0}$. In this way, identifying Integrity violations might become easier because we just need to compare the total weight of the f servers having the greatest weights with $\frac{\texttt{W}_{\mathcal{S},0}}{2}$.

To apply this restriction, servers can reassign their weights in a pairwise manner, i.e., a server s_j gains a weight Δ if and only if another server s_i loses Δ. In such a situation, we say that Δ is *transferred* from s_i to s_j. To represent this way of reassigning weights, we define a new operation as follows:

- $\texttt{transfer}(s_i, s_j, \Delta)$ that can be invoked by any server s_k to transfer $\Delta \neq 0$ from s_i to s_j.

Similar to the weight reassignment problem, processes can utilize the $\texttt{read\_changes}$ operation to learn about changes created for each server. When $\texttt{transfer}(s_i, s_j, \Delta)$ invoked by s_k is completed, two changes $c = \langle s_k, lc_k, s_i, -\Delta'\rangle$ and $c' = \langle s_k, lc_k, s_j, \Delta'\rangle$ corresponding to the transfer's outcome are created, where Δ' is either Δ or 0. Further, a message $\langle \texttt{Complete}, c\rangle$ is returned to s_k (notice that both changes

are created with either non-zero weights or zero weights, so returning c is enough to determine the weight of c'). We say that the transfer is completed if there is a time after which the responses of two invocations $\texttt{read\_changes}(s_i)$ and $\texttt{read\_changes}(s_j)$ contain c and c', respectively. Each server increments its local counter after each `transfer` invocation. By convention, we say that a `transfer` invocation is *effective* (resp. *null*) if the weights of created changes are non-zero (resp. zero).

By considering this restriction, we define a new variant of the weight reassignment problem called the *pairwise weight reassignment* in which `transfer` is the only operation to reassign weights. The definition of the pairwise weight reassignment contains all properties of the weight reassignment problem (Definition 3) adapted to use the `transfer` operation instead of `reassign`, as follows.

Definition 4 (Pairwise Weight Reassignment)**.** Any algorithm that solves the pairwise weight reassignment problem must satisfy the following properties:

- P-Integrity. $\forall t \geq 0$, $\forall F \subset \mathcal{S}$ such that $|F| = f$, $\texttt{W}_{F,t} < \frac{\texttt{W}_{\mathcal{S},t}}{2}$.
- P-Validity-I. When the $\texttt{transfer}(s_i, s_j, \Delta)$ operation is completed, two changes $\langle *, *, s_i, -\Delta\rangle$ and $\langle *, *, s_j, \Delta\rangle$ are created if P-Integrity is not violated; otherwise, two changes $\langle *, *, s_i, 0\rangle$ and $\langle *, *, s_j, 0\rangle$ are created.
- P-Validity-II. If $\texttt{read\_changes}(s)$ is invoked at time t, a set containing $\mathcal{C}_{s,t}$ is returned as the response.
- P-Liveness. If any correct server s invokes `transfer` (resp. a correct process p invokes `read_changes`), the invocation will eventually be completed, and s (resp. p) will receive a message $\langle \texttt{Complete}, *\rangle$ (resp. a set of changes).

Now this question arises: *Can the pairwise weight reassignment be implemented in asynchronous failure-prone systems?* The answer to this question is *no*. The general idea behind this impossibility is similar to the one presented for the weight reassignment problem (Section IV). Consider a set of servers $F \subset \mathcal{S}$ with size f, and assume that all correct servers invoke `transfer` concurrently such that only one of the transfers executed by members of $\mathcal{S} \setminus F$ can be completed effectively. P-Integrity is indeed violated if two or more transfers executed by members of $\mathcal{S} \setminus F$ are completed effectively. In such a situation, all correct servers can decide on the value proposed by a server $s \in \mathcal{S} \setminus F$ whose transfer is completed effectively (the decided value is selected from the values proposed by members of $\mathcal{S} \setminus F$.) As a result, servers can solve consensus using pairwise weight reassignment, which means that consensus can be reduced to the pairwise weight reassignment problem. Hence, pairwise weight reassignment cannot be implemented in asynchronous failure-prone systems.

Based on this insight, we design an algorithm, presented in Algorithm 2, to solve consensus using pairwise weight reassignment. The algorithm is executed by each correct server s_i and provides a function $\texttt{propose}(v_i)$. We assume that the initial weight of each server $s \in F$ (resp. $s \in \mathcal{S} \setminus F$) is $\frac{n-1}{2f}$ (resp. $\frac{n+1}{2(n-f)}$), where $F = \{s_1, s_2, \ldots, s_f\}$. Further, there is a shared array of SWMR registers R with size n to store servers' proposals.

Algorithm 2 Reducing consensus to the pairwise weight reassignment problem – server s_i.

▷ R is a shared array of SWMR registers with size n
▷ if $i \in \{1, 2, \ldots f\}$, $\texttt{W}_{s_i,0} = \frac{n-1}{2f}$; otherwise, $\texttt{W}_{s_i,0} = \frac{n+1}{2(n-f)}$
▷ s_i executes the `propose` function

function $\texttt{propose}(v_i)$
1: $R[i] \leftarrow v_i$
2: **if** $i \in \{1, 2, \ldots, f\}$
3: $\quad j \leftarrow (i+1)\ mod\ f$
4: $\quad \texttt{transfer}(s_i, s_j, 0.1)$
5: **else**
6: $\quad \texttt{transfer}(s_i, s_1, 0.4)$
7: $decided\_value \leftarrow \perp$
8: **repeat**
9: $\quad$ **for** $j \in \{f+1, f+2, \ldots n\}$
10: $\quad\quad$ **if** $\langle s_j, 2, s_1, 0.4\rangle \in \texttt{read\_changes}(s_j)$
11: $\quad\quad\quad decided\_value \leftarrow R[j]$
12: **until** $decided\_value \neq \perp$
13: **return** $decided\_value$

Each server $s_i \in \mathcal{S}$ executes the `propose` function. After storing its proposal in $R[i]$, each server s_i invokes `transfer`. The transfers executed by servers must be in such a way that only one of the transfers executed by members of $\mathcal{S} \setminus F$ can be completed effectively. To this end, each server $s_i \in F$ (resp. $s_i \in \mathcal{S} \setminus F$) invokes $\texttt{transfer}(s_i, s_j, 0.1)$ (resp. $\texttt{transfer}(s_i, s_1, 0.4)$), where $j = (i+1)\ mod\ f$.

In the loop, for each server $s_j \in \mathcal{S} \setminus F$, s_i repeatedly invokes $\texttt{read\_changes}(s_j)$ to see the transfer of which server is completed effectively. Because of P-Liveness, the loop will eventually terminate. Assume that the transfer of server s_j is completed effectively by creating two changes $\langle s_j, 2, s_j, -0.4\rangle$ and $\langle s_j, 2, s_1, 0.4\rangle$. Consequently, s_i returns $R[j]$ as the decided value.

It is straightforward to see that the transfer of each correct server $s \in F$ completes effectively without changing the total weight of servers in F. On the other hand, only one transfer executed by members of $\mathcal{S} \setminus F$ can be completed effectively; otherwise, P-Integrity is violated. For instance, if transfers of s_{f+1} and s_{f+2} are completed effectively by creating changes $\langle s_{f+1}, 2, s_{f+1}, -0.4\rangle$, $\langle s_{f+1}, 2, s_1, 0.4\rangle$, $\langle s_{f+2}, 2, s_{f+2}, -0.4\rangle$, and $\langle s_{f+2}, 2, s_1, 0.4\rangle$ at time $t > 0$, then we have:

$$\texttt{W}_{F,t} = f \times \frac{n-1}{2f} + 0.4 + 0.4 = \frac{n}{2} + 0.3$$
$$\frac{\texttt{W}_{\mathcal{S},t}}{2} = \frac{\texttt{W}_{\mathcal{S},0}}{2} = \frac{n}{2},$$

which means that P-Integrity is violated.

Theorem 2. Consensus can be reduced to the weight reassignment problem.

As the restriction presented above is insufficient to restrict the weight reassignment problem in a way that can be implemented in asynchronous failure-prone systems, we define another problem in the following.

B. Restricted pairwise weight reassignment

In pairwise weight reassignment, after invoking `transfer`, servers must use consensus or similar primitives to create changes in order to preserve P-Integrity. The objective of using consensus for each `transfer` invocation is to *decide* whether the invocation is effective or not, i.e., which changes must be created: two changes with non-zero weights or with zero weights. One possible approach to create changes without consensus is eliminating such a globally taken decision, i.e., given a server s_i that wants to invoke $\texttt{transfer}(*, *, \Delta)$, s_i is allowed to execute the operation if its invocation does not violate (P-)Integrity.

We now present two conditions that, if they are satisfied, ensure an effective transfer:

- (C1) only s_i can invoke $\texttt{transfer}(s_i, *, \Delta)$, i.e., other servers cannot transfer some of s_i's weight, and
- (C2) the weight of s_i must always be greater than $\frac{\mathtt{W}_{\mathcal{S},0}}{2(n-f)}$.

Note that if C1 holds, C2 is a locally verifiable condition.

Theorem 3. Provided that a server s_i wants to invoke `transfer`, we can ensure that the transfer is effective if C1 and C2 are met.

The proof of Theorem 3 can be found in the appendix. Here we present an insight into these conditions. It is clear that:

$$\mathtt{W}_{\mathcal{S},t} = \mathtt{W}_{\mathcal{S}\setminus F,t} + \mathtt{W}_{F,t} \qquad (\forall F \subset \mathcal{S}, \forall t \geq 0) \quad (1)$$

It is straightforward to obtain the following inequality using (P-)Integrity and Equation 1 when $|F| = f$:

$$\mathtt{W}_{\mathcal{S}\setminus F,t} > \frac{\mathtt{W}_{\mathcal{S},t}}{2} \quad (\forall F \subset \mathcal{S} \text{ such that } |F| = f, \forall t \geq 0) \quad (2)$$

Inequality 2, which is equivalent to (P-)Integrity, states that the total weight of any $n - f$ servers must be greater than half of the total weight of all servers. Notice that if the weight of each server is greater than $\frac{\mathtt{W}_{\mathcal{S},0}}{2(n-f)}$ at any time t, the total weight of servers in set $\mathcal{S} \setminus F$ is greater than $|\mathcal{S}\setminus F| \times \frac{\mathtt{W}_{\mathcal{S},0}}{2(n-f)} = \frac{\mathtt{W}_{\mathcal{S},0}}{2}$, i.e., $\mathtt{W}_{\mathcal{S}\setminus F,t} > \frac{\mathtt{W}_{\mathcal{S},0}}{2}$. Hence, if C2 holds for each transfer, (P-)Integrity is always preserved. To see why C1 is required, assume that at least two servers $s_i, s_k \neq s_j$ invoke $\texttt{transfer}(s_j, *, \Delta_1)$ and $\texttt{transfer}(s_j, *, \Delta_2)$ at time t such that $\mathtt{W}_{s_j,t} - \Delta_1 - \Delta_2 \leq \frac{\mathtt{W}_{\mathcal{S},0}}{2(n-f)}$ but $\mathtt{W}_{s_j,t} - \Delta_1 > \frac{\mathtt{W}_{\mathcal{S},0}}{2(n-f)}$ and $\mathtt{W}_{s_j,t} - \Delta_2 > \frac{\mathtt{W}_{\mathcal{S},0}}{2(n-f)}$. In fact, if both transfers are completed effectively, then C2 is violated; however, one of the transfers can be completed effectively without violating C2. In such a situation, s_i and s_k can solve consensus (like the impossibility results presented in Sections IV and V.) Consequently, in order to satisfy C2 in asynchronous failure-prone systems, we must assume that for each server s_j, there is at most one server that is allowed to invoke $\texttt{transfer}(s_j, *, *)$. Without loss of generality, we assume that only s_j can invoke $\texttt{transfer}(s_j, *, *)$.

These conditions indeed can restrict pairwise weight reassignment. We define a new version of pairwise weight reassignment called *restricted pairwise weight reassignment* to consider these conditions. Specifically, it contains all properties of the pairwise weight reassignment (Definition 4) except for two changes: P-Integrity is replaced by RP-Integrity to consider C2, and P-Validity-I is adapted so that only server s can invoke $\texttt{transfer}(s, *, *)$ due to C1. In the next section, we elaborate on how servers can use these conditions to transfer weights in asynchronous failure-prone systems.

Definition 5 (Restricted Pairwise Weight Reassignment)**.** Any algorithm that solves the restricted pairwise weight reassignment problem must satisfy the following properties:

- RP-Integrity. $\forall t \geq 0$, $\forall s \in \mathcal{S}$, $\mathtt{W}_{s,t} > \frac{\mathtt{W}_{\mathcal{S},0}}{2(n-f)}$.
- RP-Validity-I. When the $\texttt{transfer}(s_i, s_j, \Delta)$ operation is completed, two changes $\langle s_i, *, s_i, -\Delta\rangle$ and $\langle s_i, *, s_j, \Delta\rangle$ are created if RP-Integrity is not violated; otherwise, two changes $\langle s_i, *, s_i, 0\rangle$ and $\langle s_i, *, s_j, 0\rangle$ are created.
- RP-Validity-II. If $\texttt{read\_changes}(s)$ is invoked at time t, a set containing $\mathcal{C}_{s,t}$ is returned as the response.
- RP-Liveness. If any correct server s invokes `transfer` (resp. a correct process p invokes `read_changes`), the invocation will eventually be completed, and s (resp. p) will receive a message $\langle \texttt{Complete}, *\rangle$ (resp. a set of changes).

Example 2. Let $\mathcal{S} = \{s_1, s_2, s_3, s_4, s_5, s_6, s_7\}$ and $f = 2$. In this setting, the weight of each server must be greater than 0.7 at any time $t \geq 0$. Notice that the size of each quorum is four at the beginning. Assume that `transfer` is invoked by s_4, s_5, and s_6 according to Fig. 1, and the invocations are completed before time t_1. The new weights of servers at time t_1 are presented in the figure. As a result of these weight reassignments, $\{s_1, s_2, s_3\}$ (a minority of servers) constitutes a quorum.

This figure contains two other invocations made by s_6 and s_7 after time t_1. Notice that these invocations cannot be executed in the restricted pairwise weight reassignment due to RP-Integrity violation. However, they could be executed in the pairwise weight reassignment, resulting in the weights in the red shaded rectangular area.

C. Discussion

The restrictions imposed by the pairwise weight reassignment and restricted pairwise weight reassignment problems can lead to practical limitations in dealing with failed or slow servers in asynchronous systems. Here, we discuss these limitations in further detail.

In the weight reassignment problem, in the case of having a failed/slow server, there are two possible approaches to mitigate the impact of such a server: (I) decreasing the weight of the failed/slow server by other servers or (II) increasing the weights of other servers. This flexibility allows other servers to form quorums by a minority of servers during execution.

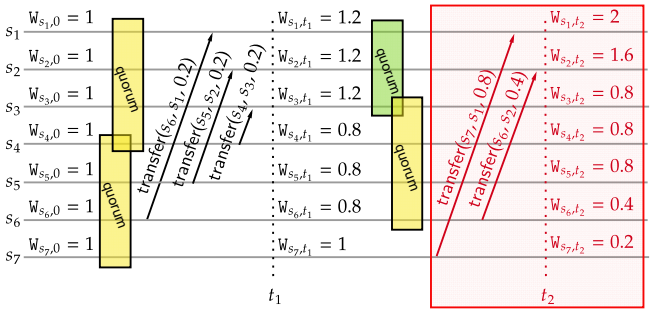

Fig. 1: An example showing how the restricted pairwise weight reassignment works. The part surrounded by a red box cannot be executed in the restricted pairwise weight reassignment.

However, in pairwise weight reassignment, the second approach cannot be employed in the case of having a failed/slow server, as the total weight of servers cannot change. The situation is even worse for the restricted pairwise weight reassignment, as servers cannot use both approaches when having a failed/slow server. For instance, consider the same system as presented in Example 2. Assume that the initial weights of servers $s_1, s_2, s_3, s_4, s_5, s_6$, and s_7 are $1.6, 1.4, 0.8, 0.8, 0.8, 0.8$, and 0.8, respectively. Assume that servers s_1 and s_2 are failed or slow. Then, the size of the smallest quorum is five, and servers cannot form smaller quorums by reassigning weights.

VI. Implementing the Restricted Pairwise Weight Reassignment

This section presents a protocol that implements the restricted pairwise weight reassignment in asynchronous failure-prone systems. The protocol is composed of two algorithms. The first algorithm, Algorithm 3, implements the `read_changes` operation and contains two parts: the first part can be executed by any process (lines 1-9) and the second part must be executed by each correct server (lines 12-15). To read the changes created for a server s, a process p_i invokes `read_changes`. Then, p_i broadcasts a message $\langle \texttt{RC}, s, lc_i \rangle$ to all servers. Upon receiving the message, each server responds by sending a set of changes it has stored for server s. When p_i receives more than f responses, it takes the union of the received set of changes. Let $\mathcal{C}$ be the resulting set. Then, p_i broadcasts $\mathcal{C}$ to all servers. After receiving $\mathcal{C}$, each server stores $\mathcal{C}$ and responds by sending an acknowledgment. As soon as receiving $n-f$ acknowledgments, p can ensure that $\mathcal{C}$ is stored by at least $n-f$ servers and completes the invocation by returning $\mathcal{C}$.

The second algorithm, Algorithm 4, contains the pseudo-code of the `transfer` operation and must be executed by each correct server. It is worth mentioning that a server can invoke `transfer` only if its last invocation of `transfer` is complete. We assume that servers invoke `transfer` based on the information provided by a monitoring system (to see how it can be implemented or its provided information can be used, refer to [10], [11]).

Algorithm 3 The implementation of `read_changes`.

▷ This part can be executed by all process.
operation `read_changes`(s)
1: $\mathcal{C} \leftarrow \emptyset$
2: **broadcast** $\langle \texttt{RC}, s \rangle$
3: **repeat**
4: **wait until** $\langle \texttt{RC\_Ack}, \mathcal{C}_s \rangle$ is received from a server
5: $\mathcal{C} \leftarrow \mathcal{C} \cup \mathcal{C}_s$
6: **until** $|\mathcal{C}| > f$
7: **broadcast** $\langle \texttt{WC}, \mathcal{C} \rangle$
8: **wait until** $\langle \texttt{WC\_Ack} \rangle$ is received from $n - f$ servers
9: **return** $\mathcal{C}$

▷ This part is executed by every correct server s_i.
upon receipt of $\langle \texttt{RC}, s \rangle$ from p
12: $\mathcal{C} \leftarrow$ `get_changes`(s) ▷ see Algorithm 4
13: **send** $\langle \texttt{RC\_Ack}, \mathcal{C} \rangle$ to p

upon receipt of $\langle \texttt{WC}, \mathcal{C} \rangle$ from p
14: `write_changes`$(\mathcal{C})$ ▷ see Algorithm 4
15: **send** $\langle \texttt{WC\_Ack} \rangle$ to p

The main idea behind Algorithm 4 is as follows. Each server s can compute its weight using the changes stored in a local variable. If its weight is greater than $\Delta + \frac{W_{S,0}}{2(n-f)}$, it can transfer Δ to another server.

In further detail, each server has a local counter, denoted by lc, initialized with 1, and used to distinguish the transferred weights. Also, each server has a variable denoted by *register* to store the tag and the value of its local register (we elaborate on this variable in the next section.) If a server s_i wants to transfer weight Δ to another server s_j, it first examines whether its weight remains greater than $\frac{W_{S,0}}{2(n-f)}$ by transferring Δ (line 12). If this is the case, then s_i broadcasts a message $\langle \texttt{T}, \langle s_i, lc_i, s_i, -\Delta \rangle, \langle s_i, lc_i, s_j, \Delta \rangle \rangle$ using a reliable broadcast primitive [25] (line 14). Each server has a set denoted by $\mathcal{C}$ to store every received change, initialized with a set containing the initial weights of all servers. By receiving a message $\langle \texttt{T}, c, c' \rangle$ broadcast by s_i (line 21), every server adds c and c' to its set $\mathcal{C}$ and sends a response to s_i (lines 10-11). If s_i receives at least $n-f$ responses, then the invocation completes (lines 15-20).

Theorem 4. The implementation of restricted pairwise weight reassignment (Algorithms 3 and 4) satisfies the properties of the problem (Definition 5).

Theorem 5. Restricted pairwise weight reassignment can be implemented in asynchronous failure-prone systems.

VII. Dynamic-Weighted Atomic Storage

This section demonstrates how storage systems based on MQS in asynchronous systems can be adapted to leverage the advantages of dynamic WMQS. To do so, we construct a dynamic-weighted atomic storage, where the weights of servers can be reassigned using the restricted pairwise weight

Algorithm 4 The implementation of the `transfer` operation – server s_i.

variables
1: $lc_i \leftarrow 1$
2: $\mathcal{C} \leftarrow \{\langle s, 1, s, 1\rangle \mid \forall s \in \mathcal{S}\}$
3: $register[tag[ts, pid], val] \leftarrow \langle\langle 0, \perp\rangle, \perp\rangle$

function `weight`()
4: $T \leftarrow$ `get_changes`(s_i)
5: **return** $\sum_{\langle *,*,*,\Delta\rangle \in T} \Delta$

function `get_changes`(s)
6: **return** $\{\langle *, *, s', *\rangle \mid \forall\ \langle *, *, s', *\rangle \in \mathcal{C} : s' = s\}$

function `write_changes`$(\mathcal{C}')$
7: **for all** $\langle s_j, c, s_k, *\rangle \in \mathcal{C}' \setminus \mathcal{C}$
8: **if** $i = k$
9: $register \leftarrow$ `read`()
10: $\mathcal{C} \leftarrow \mathcal{C} \cup \{\langle s_j, c, s_k, *\rangle\}$
11: **send** $\langle$`T_Ack`$, c\rangle$ to s_j if not already sent

operation `transfer`(s_i, s_j, Δ)
12: **if** `weight`$() > \Delta + \frac{W_{S,0}}{2(n-f)}$
13: $\mathcal{C} \leftarrow \mathcal{C} \cup \{\langle s_i, lc_i, s_i, -\Delta\rangle, \langle s_i, lc_i, s_j, \Delta\rangle\}$
14: **RB_broadcast** $\langle$`T`$, \langle s_i, lc_i, s_i, -\Delta\rangle, \langle s_i, lc_i, s_j, \Delta\rangle\rangle$
15: **wait until** receiving $\langle$`T_Ack`$, lc_i\rangle$ from $n-f-1$ servers
16: $msg \leftarrow \langle$`Complete`$, \langle s_i, lc_i, s_i, -\Delta\rangle\rangle$
17: **else**
18: $msg \leftarrow \langle$`Complete`$, \langle s_i, lc_i, s_i, 0\rangle\rangle$
19: $lc_i \leftarrow lc_i + 1$
20: **return** msg

upon RB_deliver $\langle$`T`$, \langle s_j, c, s_j, -\Delta\rangle, \langle s_j, c, s_k, \Delta\rangle\rangle$
21: `write_changes` $(\{\langle s_j, c, s_j, -\Delta\rangle, \langle s_j, c, s_k, \Delta\rangle\})$

reassignment, and its stored value can be accessed by two operations: `read` and `write`.

In a nutshell, there are two main requirements to construct such storage. First, each process p (client or server) that wants to execute `read` or `write` protocols needs to store the most up-to-date set of the completed changes $\mathcal{C}$ that it knows. All `read`/`write` protocol messages carry $\mathcal{C}$, and p updates it as soon as discovering a more up-to-date set of completed changes. The servers reject any operation issued by p containing a set of changes different from $\mathcal{C}$ and respond by sending their current set of completed changes to p, which updates its set $\mathcal{C}$. By receiving a set of changes that differs from its set of changes, p restarts the executing operation. The second requirement is that, before accessing the system, p must know the initial weight of each server.

Our protocol extends the classical ABD algorithm [26] for supporting multiple writers and working with the restricted pairwise weight reassignment. In the following, we highlight the main aspects of the `read` and `write` protocols (the complete algorithms can be found in the appendix.) These protocols work in phases. Each phase corresponds to accessing a weighted quorum of servers in $\mathcal{C}$. The `read` protocol works as follows:

- *1st Phase:* a reader requests a set of tuples $\langle tag, val\rangle$ (val is the value a server stores, and tag is its associated tag[3]) from a weighted quorum of servers in the most up-to-date set of completed changes $\mathcal{C}$ and selects the one with the highest tag $\langle tag_h, val_h\rangle$;
- *2nd Phase:* the reader performs an additional write-back phase in the system and waits for confirmations from a weighted quorum of servers in $\mathcal{C}$ before returning val_h.

The `write` protocol works in a similar way:

- *1st Phase:* a writer obtains a set of tags from a weighted quorum of servers in $\mathcal{C}$ and chooses the highest, tag_h; the tag to be written is defined by incrementing $tag_h.ts$ and assigning $tag_h.pid$ to the writer's id;
- *2nd Phase:* the writer sends a tuple $\langle tag, val\rangle$ to the servers of $\mathcal{C}$, writing val with tag tag, and waits for confirmations from a weighted quorum.

Correctness. The following theorem states that the dynamic-weighted atomic storage can be implemented using the `read`/`write` protocols if the weights of servers are reassigned by invoking `transfer` (Algorithm 4).

Theorem 6. The storage system implemented using the described `read`/`write` protocols is atomic storage.

VIII. Related Work

Weight (re)assignment. The notion of *majority* quorum was extended to be a *weighted majority* quorum to improve the performance of replicated systems with diverse servers assigned with different voting power [19]. WHEAT (WeighHt-Enabled Active replicaTion) [20] shows that additional spare servers and weighted voting allow the system to benefit from diverse quorum sizes, enabling it to make progress by employing proportionally smaller quorums and potentially obtaining significant latency improvements. In WHEAT, each assigned weight is either w_{min} or w_{max}. These values are defined in such a way that the safety and liveness properties of quorums are always satisfied. We also considered a minimum weight for defining the restricted pairwise weight reassignment, just like WHEAT. Still, we consider more general weight schemes since our weights can have any value greater than the defined minimum.

In practice, network characteristics may be subject to run-time variations, and thus the assigned weights may also require to be changed over time. Accordingly, the problem of integrating WMQS with weight reassignment protocols was introduced to allow reassigning weights over time according to the observed performance variations. This problem was studied for partially synchronous systems in [10], [20], [22], [27], [28], where the weight reassignment protocols are based on consensus or similar primitives. Besides, such a problem was studied for asynchronous systems in [11], where the weights

[3] A tag tg is a pair holding the timestamp $tg.ts$ and the writer's id $tg.pid$ associated with the value stored by the register. A tag tg_1 is less than another tg_2 if (I) $tg_1.ts < tg_2.ts$, or (II) $tg_1.ts = tg_2.ts$ and $tg_1.pid < tg_2.pid$.

can be reassigned in a pairwise manner using an epoch-based protocol. In the presented protocol, reassignment requests issued during an epoch can only be applied at the end of the epoch. Notice that the duration of epochs impacts the performance of the protocol, and determining the optimal duration for epochs is challenging. That study inspires our restricted pairwise weight reassignment; however, our implementation is epochless. Moreover, in that study, the total weight of servers might become less than $\mathtt{W}_{\mathcal{S},0}$, leading to the loss of voting power as the system progresses.

Relationship with the asset transfer problem. In the asset transfer problem, there are some accounts, each holding some assets owned by $k \geq 1$ servers. Some assets of any account can be transferred to another account if the source's balance does not become negative. It was proved by Guerraoui et al. [12] that if there is an account with $k > 1$ owners, the consensus number of the problem is k, i.e., the problem cannot be implemented in asynchronous failure-prone systems. The insight into such an impossibility is as follows. Consider an account with $k > 1$ owner, and assume that all owners concurrently want to transfer some assets from the account to another account(s) such that the balance of the account will become negative by executing all transfers. In such a situation, some transfers must be aborted to keep the account's balance non-negative, which requires consensus.

Reassigning weights in a pairwise manner is inspired by the asset transfer problem (consider weights equivalents to assets.) These problems are similar in reassigning/transferring weights/assets, and both cannot be implemented in asynchronous failure-prone systems. However, there is a significant difference between them: there is no condition related to the distribution of assets in the asset transfer problem, but the total of the f greatest weights should be less than half of the total weights in the pairwise weight reassignment to satisfy P-Integrity.

To solve the asset transfer problem in asynchronous failure-prone systems, a restricted version of the problem called the 1-asset transfer problem is presented in which each account is owned by exactly one owner. It was proved that such a restricted problem could be implemented in asynchronous failure-prone systems [12]. In the restricted pairwise weight reassignment, the assumption that transferring a weight Δ from a server s to another server can be made only by s is inspired by the 1-asset transfer problem.

Relationship with asynchronous reconfiguration. Reconfigurable atomic storage [13]–[17] implements atomic registers [29] in systems with the possibility of changing the set of servers over time, i.e., servers can join and leave the system during an execution. The reconfigurable atomic storage is similar to the dynamic-weighted atomic storage in which quorum formations might change over time, i.e., a subset of servers that form a quorum during a time interval might not form a quorum after that interval. Based on such a similarity, one might say that the techniques used to solve the reconfigurable atomic storage, e.g., generalized lattice agreement [18], can be employed to solve the dynamic-weighted atomic storage; however, this is not the case because the system's availability is defined differently in these problems.

In dynamic-weighted atomic storage, the system remains available as long as the number of failures does not exceed the fault threshold; however, in reconfigurable atomic storage, the system remains available as long as any pending configuration has a majority of servers that did not crash and were not proposed for removal. In other words, the fault threshold is static and independent of the reassignment requests in the dynamic-weighted atomic storage; however, the fault threshold is dynamic and determined based on the pending join and leave requests in reconfigurable atomic storage (see Definition 1 in [13]).

IX. Conclusion

This paper studies the problem of integrating weighted majority quorums with weight reassignment protocols for any asynchronous system with a static set of servers and static fault threshold while guaranteeing availability. We showed that such a problem could not be solved in asynchronous failure-prone distributed systems. Then, we presented a restricted version of the problem called *pairwise weight reassignment*, in which weights can only be reassigned pairwisely. We showed that pairwise weight reassignment could not be implemented in asynchronous failure-prone systems. We also discussed the relation between the pairwise weight reassignment and the asset transfer problem. We presented a restricted version of the pairwise weight reassignment called *restricted pairwise weight reassignment* that can be implemented in asynchronous failure-prone systems. As a case study, we presented a dynamic-weighted atomic storage based on the implementation of the restricted pairwise weight reassignment.

Acknowledgments

We thank the ICDCS'23 anonymous reviewers for their constructive comments to improve the paper. This work was supported by the Ministry of Higher Education and Research of France, FCT through the ThreatAdapt project (FCT-FNR/0002/2018) and the LASIGE Research Unit (UIDB/00408/2020 and UIDP/00408/2020), and by the European Commission through the VEDLIoT project (H2020 957197).

References

[1] M. Vukolić, *Quorum Systems: With Applications to Storage and Consensus*. Morgan & Claypool, 2012.
[2] M. Naor and A. Wool, "The load, capacity, and availability of quorum systems," *SIAM Journal on Computing*, vol. 27, no. 2, 1998.
[3] D. Agrawal and A. El Abbadi, "The tree quorum protocol: An efficient approach for managing replicated data." in *16th International Conference on Very Large Data Bases*, 1990.
[4] "Etcd," https://github.com/etcd-io/etcd, accessed: 2022-07-05.
[5] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free coordination for internet-scale systems." in *USENIX annual technical conference*, 2010.
[6] Y. Saito, S. Frølund, A. Veitch, A. Merchant, and S. Spence, "FAB: building distributed enterprise disk arrays from commodity components," *ACM SIGOPS Operating Systems Review*, vol. 39, no. 5, 2004.

[7] F. B. Schmuck and R. L. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, 2002.
[8] M. Whittaker, A. Charapko, J. M. Hellerstein, H. Howard, and I. Stoica, "Read-write quorum systems made practical," in *Proceedings of the 8th Workshop on Principles and Practice of Consistency for Distributed Data*, 2021.
[9] D. Barbara and H. Garcia-Molina, "The reliability of voting mechanisms," *IEEE Transactions on Computers*, vol. 36, no. 10, 1987.
[10] C. Berger, H. P. Reiser, J. Sousa, and A. Neves Bessani, "AWARE: Adaptive wide-area replication for fast and resilient Byzantine consensus," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 3, 2022.
[11] H. Heydari, G. Silvestre, and L. Arantes, "Efficient consensus-free weight reassignment for atomic storage," in *IEEE 20th International Symposium on Network Computing and Applications*, 2021.
[12] R. Guerraoui, P. Kuznetsov, M. Monti, M. Pavlovič, and D.-A. Seredinschi, "The consensus number of a cryptocurrency," in *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, 2019.
[13] M. K. Aguilera, I. Keidar, D. Malkhi, and A. Shraer, "Dynamic atomic storage without consensus," *Journal of the ACM*, vol. 58, no. 2, 2011.
[14] E. Alchieri, A. Bessani, F. Greve, and J. Fraga, "Efficient and modular consensus-free reconfiguration for fault-tolerant storage," in *21st International Conference on Principles of Distributed Systems*, 2017.
[15] L. Jehl and H. Meling, "The case for reconfiguration without consensus: Comparing algorithms for atomic storage," in *21st International Conference on Principles of Distributed Systems*, 2017.
[16] L. Jehl, R. Vitenberg, and H. Meling, "SmartMerge: A new approach to reconfiguration for atomic storage," in *International Symposium on Distributed Computing*, 2015.
[17] A. Spiegelman, I. Keidar, and D. Malkhi, "Dynamic reconfiguration: Abstraction and optimal asynchronous solution," in *International Symposium on Distributed Computing*, 2017.
[18] J. M. Faleiro, S. Rajamani, K. Rajan, G. Ramalingam, and K. Vaswani, "Generalized lattice agreement," in *Proceedings of the 2012 ACM symposium on Principles of distributed computing*, 2012.
[19] D. K. Gifford, "Weighted voting for replicated data," *Proceedings of the seventh ACM symposium on Operating systems*, 1979.
[20] J. Sousa and A. Bessani, "Separating the wheat from the chaff: An empirical design for geo-replicated state machines," in *IEEE 34th Symposium on Reliable Distributed Systems*, 2015.
[21] H. Garcia-Molina and D. Barbara, "How to assign votes in a distributed system," *Journal of the ACM*, vol. 32, no. 4, 1985.
[22] S. Jajodia and D. Mutchler, "Dynamic voting algorithms for maintaining the consistency of a replicated database," *ACM Transactions on Database Systems*, vol. 15, no. 2, 1990.
[23] M. J. Fischer, "The consensus problem in unreliable distributed systems (a brief survey)," in *International Conference on Fundamentals of Computation Theory*, 1983.
[24] M. Herlihy, "Wait-free synchronization," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 1, 1991.
[25] V. Hadzilacos and S. Toueg, "A modular approach to fault-tolerant broadcasts and related problems," Cornell University, USA, Tech. Rep. TR94-1425, 5 1994.
[26] H. Attiya, A. Bar-Noy, and D. Dolev, "Sharing memory robustly in message-passing systems," *Journal of the ACM*, vol. 42, no. 1, 1995.
[27] C. Berger, L. Rodrigues, H. P. Reiser, V. Cogo, and A. Bessani, "Chasing the speed of light: Low-latency planetary-scale adaptive Byzantine consensus," https://arxiv.org/abs/2305.15000, 2023.
[28] D. Davcev, "A dynamic voting scheme in distributed systems," *IEEE Transactions on Software Engineering*, vol. 15, no. 1, 1989.
[29] L. Lamport, "On interprocess communication (part I)," *Distributed Computing*, vol. 1, no. 2, 1986.
[30] ——, "On interprocess communication (part II)," *Distributed Computing*, vol. 1, no. 2, 1986.
[31] P. Kuznetsov, T. Rieutord, and S. Tucci-Piergiovanni, "Reconfigurable Lattice Agreement and Applications," in *23rd International Conference on Principles of Distributed Systems*, 2020.
[32] A. Spiegelman and I. Keidar, "On liveness of dynamic storage," in *International Colloquium on Structural Information and Communication Complexity*, 2017.

APPENDIX

Proof of Theorem 1. We show that servers can solve consensus using Algorithm 1, i.e., three properties of consensus – Agreement, Validity, and Termination – can be satisfied.

- (Agreement) Recall that each server s executes the `propose` function in Algorithm 1. Then, s invokes `reassign` according to its identifier. We first show that only one of the `reassign` invocations can be completed by creating a change with non-zero weight. For the sake of contradiction, assume that there is a set $A \subseteq \mathcal{S}$ such that $|A| \geq 2$ and the invocation of each server $s \in A$ completes at time $t > 0$ by creating a change with non-zero weight. Let $F = \{s_1, \ldots, s_f\}$. Assume that k members of A are in F, i.e., $|A \cap F| = k$. We have:

$$\mathtt{W}_{F,t} = \underbrace{f \times \frac{n-1}{2f}}_{\mathtt{W}_{F,0}} + k \times 0.5 \tag{3}$$

$$\mathtt{W}_{\mathcal{S}\setminus F,t} = \underbrace{(n-f) \times \frac{n+1}{2(n-f)}}_{\mathtt{W}_{\mathcal{S}\setminus F,0}} - (|A| - k) \times 0.5$$

  It is straightforward to obtain the following inequality using Integrity and Inequality 2:

$$\mathtt{W}_{F,t'} < \mathtt{W}_{\mathcal{S}\setminus F,t'} (\forall F \subset \mathcal{S} \text{ such that } |F| = f, \forall t' \geq 0) \tag{4}$$

  From Equations 3 and Inequality 4, we have:

$$\begin{aligned} &\mathtt{W}_{F,t'} < \mathtt{W}_{\mathcal{S}\setminus F,t'} \\ &\quad \Rightarrow \frac{n-1}{2} + k \times 0.5 < \frac{n+1}{2} - (|A| - k) \times 0.5 \\ &\quad \Rightarrow |A| < 2, \end{aligned}$$

  which is a contradiction since we assumed that $|A| \geq 2$. Next, we show that all invocations cannot be completed by creating changes with zero weights. For contradiction, assume all invocations are completed by creating changes with zero weights. There are two possibilities for a correct server s_i: $s_i \in F$ or $s_i \in \mathcal{S} \setminus F$. If $s_i \in F$, the invocation $\mathtt{reassign}(s_i, 0.5)$ could be completed by creating a change with weight 0.5, as Integrity is still preserved. Likewise, if $s_i \in \mathcal{S} \setminus F$, the invocation $\mathtt{reassign}(s_i, -0.5)$ could be completed by creating a change with weight -0.5. Since the invocation is completed by creating a change with zero weight, Validity-I is violated.
  Consequently, only one of the `reassign` invocations can be completed by creating a change with non-zero weight. Since the decided value corresponds to the invocation completed by creating a change with non-zero weight, the Agreement property is satisfied.
- (Validity) We say that a server s is *correct* if its `reassign` invocation is completed in Algorithm 1. We must show that if all correct servers propose the same value v, they must decide v. Note that the decided value in Algorithm 1 is among the values proposed by servers whose `reassign`

invocations are completed. In other words, the decided value is among the values proposed by the correct servers. Therefore, the Validity property is satisfied.

- (Termination) Notice that every invocation of the `reassign` operation will eventually terminate due to the Liveness property of the weight reassignment problem. Hence, the `reassign` invocation completed by creating a change with non-zero weight will eventually terminate. Also, the `read_changes` invocations that enable servers to learn which `reassign` invocation is completed by creating a change with non-zero weight will eventually terminate. Consequently, each correct server can decide eventually.

Proof of Theorem 2. Like Theorem 1, we need to show that three properties of consensus – Agreement, Validity, and Termination – can be satisfied using Algorithm 2. Let $F = \{s_1, \ldots, s_f\}$.

- (Agreement) Recall that each server $s_i \in \mathcal{S}$ executes the `propose` function in Algorithm 2. Then, s_i invokes `transfer`. Note that each server $s \in F$ transfers some of its weight to another member of F. Consequently, the total weight of members of F does not change by completing transfers executed by members of F. We now show that only one of the transfers executed by members of $\mathcal{S} \setminus F$ can be completed by creating a change with non-zero weight.
  Like the proof of Theorem 1, we first show that multiple transfers executed by members of $\mathcal{S} \setminus F$ cannot be completed by creating changes with non-zero weights. For the sake of contradiction, assume that there is a set $A \subseteq \mathcal{S} \setminus F$ such that $|A| \geq 2$ and the transfer of each server $s \in A$ completes at time $t > 0$ by creating a change with non-zero weight. We know that:

$$\mathtt{W}_{F,t} = \underbrace{f \times \frac{n-1}{2f}}_{\mathtt{W}_{F,0}} + |A| \times 0.4 \tag{5}$$

$$\mathtt{W}_{\mathcal{S}\setminus F,t} = \underbrace{(n-f) \times \frac{n+1}{2(n-f)}}_{\mathtt{W}_{\mathcal{S}\setminus F,0}} - |A| \times 0.4$$

  From Inequality 4 and Equations 5, we have:

$$\begin{aligned} &\mathtt{W}_{F,t} < \mathtt{W}_{\mathcal{S}\setminus F,t} \\ &\Rightarrow \frac{n-1}{2} + |A| \times 0.4 < \frac{n+1}{2} - |A| \times 0.4 \\ &\Rightarrow |A| < \frac{5}{4}, \end{aligned}$$

  which is a contradiction because $|A| \geq 2$ according to our assumption.
  Next, we show that all transfers executed by members of $\mathcal{S} \setminus F$ cannot be completed by creating changes with zero weights. For contradiction, assume all such transfers are completed by creating changes with zero weights. Consider a correct server $s_i \in \mathcal{S} \setminus F$. The invocation `transfer`$(s_i, s_1, 0.4)$ could be completed by creating two changes with weights 0.4 and -0.4, because by creating such changes, P-Integrity is still preserved. Since the invocation is completed by creating changes with zero weights, P-Validity-I is violated.
  Consequently, only one of the transfers executed by members of $\mathcal{S} \setminus F$ can be completed by creating two changes with non-zero weights. Since the decided value corresponds to the transfer completed by creating two changes with non-zero weights and executed by a member of $\mathcal{S} \setminus F$, the Agreement property is satisfied.
- (Validity) This property holds by the same argument presented for the Validity property in the proof of Theorem 1.
- (Termination) Every `read_changes` or `transfer` invocation will eventually terminate according to P-Liveness. Hence, the transfer executed by a member of $\mathcal{S} \setminus F$ and completed by creating a change with non-zero weight will eventually terminate. Besides, the `read_changes` invocations that enable servers to learn the transfer of which member of $\mathcal{S} \setminus F$ is completed by creating a change with non-zero weight will eventually terminate. Consequently, each correct server s can decide eventually.

Proof of Theorem 3. We present two preliminary lemmas before proving Theorem 3.

Lemma 1. If $\mathtt{W}_{s,t} > \frac{\mathtt{W}_{\mathcal{S},0}}{2(n-f)}$ for each server s at any time t, then P-Integrity is always met.

Proof. Recall that in the pairwise weight reassignment, the total weight of servers does not change during an execution, i.e., $\mathtt{W}_{\mathcal{S},t} = \mathtt{W}_{\mathcal{S},0}$ at any time $t > 0$. Also, recall that P-Integrity is equivalent to Inequality 2. Accordingly, we need to show that Inequality 2 holds if $\mathtt{W}_{s,t} > \frac{\mathtt{W}_{\mathcal{S},0}}{2(n-f)}$ for each server s at any time t. We have:

$$\begin{aligned} \mathtt{W}_{\mathcal{S}\setminus F,t} &= \sum_{s \in \mathcal{S}\setminus F} \mathtt{W}_{s,t} > \sum_{s \in \mathcal{S}\setminus F} \frac{\mathtt{W}_{\mathcal{S},0}}{2(n-f)} \\ &= |\mathcal{S} \setminus F| \times \frac{\mathtt{W}_{\mathcal{S},0}}{2(n-f)} = (n-f) \times \frac{\mathtt{W}_{\mathcal{S},0}}{2(n-f)} \\ &= \frac{\mathtt{W}_{\mathcal{S},0}}{2}, \end{aligned}$$

which means that Inequality 2 holds. □

Lemma 2. For each server s, there is at most one server that is allowed to invoke `transfer`$(s, *, *)$ in order to preserve $\mathtt{W}_{s,t} > \frac{\mathtt{W}_{\mathcal{S},0}}{2(n-f)}$ at any time t in asynchronous systems.

Proof. This proof is similar to the proof of Theorem 1. For contradiction, assume that $\mathtt{W}_{s,t} > \frac{\mathtt{W}_{\mathcal{S},0}}{2(n-f)}$ can be preserved at any time t in asynchronous systems even if multiple servers invoke `transfer`$(s, *, *)$. Particularly, assume that two correct servers $s_i, s_k \neq s_j$ invoke `transfer`$(s_j, *, \Delta_1)$ and `transfer`$(s_j, *, \Delta_2)$ at time t such that $\mathtt{W}_{s_j,t} - \Delta_1 - \Delta_2 \leq \frac{\mathtt{W}_{\mathcal{S},0}}{2(n-f)}$ but $\mathtt{W}_{s_j,t} - \Delta_1 > \frac{\mathtt{W}_{\mathcal{S},0}}{2(n-f)}$ and $\mathtt{W}_{s_j,t} - \Delta_2 > \frac{\mathtt{W}_{\mathcal{S},0}}{2(n-f)}$. This means that only one of the transfers can be completed effectively, as if both transfers are completed effectively at time $t' > t$, then $\mathtt{W}_{s,t'} \leq \frac{\mathtt{W}_{\mathcal{S},0}}{2(n-f)}$. Therefore, s_i and s_k can decide on the value proposed by a server $s \in \{s_i, s_k\}$ that its transfer is completed effectively, which is a contradiction since consensus

cannot be solved in asynchronous systems. Consequently, in order to preserve $\mathtt{W}_{s,t} > \frac{\mathtt{W}_{\mathcal{S},0}}{2(n-f)}$ in asynchronous systems, we must assume that for each server s_j, there is at most one server that is allowed to invoke $\mathtt{transfer}(s_j, *, *)$. □

Without loss of generality, we assume that only s_j can invoke $\mathtt{transfer}(s_j, *, *)$ in Lemma 2. Using Lemmas 1 and 2, the proof of Theorem 3 is immediate.

Proof of Theorem 4. We show that Algorithms 3 and 4 satisfy all properties of the restricted pairwise weight reassignment.

- (RP-Integrity) Consider a server s with a weight greater than $\Delta + \frac{\mathtt{W}_{\mathcal{S},0}}{2(n-f)}$. Assume that s invokes $\mathtt{transfer}(s, *, \Delta)$ at time $t > 0$. Since processes are sequential, it follows that s cannot have any incomplete transfer at that time. When the transfer completes, s adds a change $\langle s, lc, s, -\Delta\rangle$ to its local set $\mathcal{C}$, that contains the completed changes. As a consequence of adding such a change to $\mathcal{C}$, the weight of s decreases by Δ. Notice that before the transfer, the weight of s was greater than $\Delta + \frac{\mathtt{W}_{\mathcal{S},0}}{2(n-f)}$, so its weight remains greater than $\frac{\mathtt{W}_{\mathcal{S},0}}{2(n-f)}$ when the transfer is completed. Since only s can decrease its weight, it follows that RP-Integrity is always satisfied.
- (RP-Validity-I) Using Algorithm 4, a server s can invoke $\mathtt{transfer}(s, *, \Delta)$ only when its weight is greater than $\Delta + \frac{\mathtt{W}_{\mathcal{S},0}}{2(n-f)}$. If it is the case, two changes will be created: $\langle s, lc, s, -\Delta\rangle$ and $\langle s, lc, *, \Delta\rangle$, and a message $\langle \mathtt{Complete}, \langle s, lc, s, -\Delta\rangle\rangle$ will be returned. Otherwise, a message $\langle \mathtt{Complete}, \langle s, lc, s, 0\rangle\rangle$ will be returned without creating any change (zero-weight changes do not change the weights of servers, so it is not required to store them.) Consequently, RP-Validity-I is preserved.
- (RP-Validity-II) Assume that $\mathtt{read\_changes}(s)$ is invoked by a process p (Algorithm 3), where s is a server, and p receives a set $\mathcal{C}$ at time t as the returned value. Further, assume that there is a change $c \in \mathcal{C}$ that is completed, i.e., $c \in \mathcal{C}_{s,t}$. We need to show that if any process q that invokes $\mathtt{read\_changes}(s)$ after time t will receive a set that contains c.
  For the sake of contradiction, assume that q invokes $\mathtt{read\_changes}(s)$ after time t and receives a set $\mathcal{C}'$ that does not contain c. When q invokes $\mathtt{read\_changes}(s)$, the changes stored by at least $f+1$ servers must be collected (lines 3-6). Since $c \notin \mathcal{C}'$, it is not stored by at least $f+1$ servers. However, we know that before returning a value, it must be stored by at least $n-f$ servers in Algorithm 3 (line 8), and since $c \in \mathcal{C}$, it is stored by at least $n-f$ servers, which is a contradiction because there are n servers in the system.
- (RP-Liveness) Since at most f servers might fail, there are $n-f$ correct servers in the worst case. Note that Algorithms 3 and 4 require no more than $n-f$ correct servers. Consequently, RP-Liveness holds.

Proof of Theorem 5. According to Theorem 4, Algorithms 3 and 4 implement restricted pairwise weight reassignment. Both of these algorithms use primitives that can be implemented in asynchronous systems. Also, operations in those algorithms require to contact with at most $n-f$ servers. Since there are $n-f$ correct servers during an execution, any operation will eventually terminate. Besides, Theorem 6 shows that any invocation of `read` operation will terminate eventually. Consequently, any operation (`transfer` or `read_changes`) executed by a process can terminate eventually without waiting for the responses of other invocations of `transfer` or `read_changes` operations, meaning that both Algorithms 3 and 4 can be implemented in asynchronous failure-prone systems. It follows that restricted pairwise weight reassignment can be implemented in asynchronous failure-prone systems.

Reader-writer side of dynamic-weighted atomic storage. Algorithm 5 is the pseudo-code of the `read`/`write` protocols. The `read`/`write` protocols is similar to the `read`/`write` protocols of the ABD protocol [26] with only one difference: each reader or writer, after receiving messages from a set $Q \subseteq \mathcal{S}$ to decide whether a quorum is constituted, calls function `is_quorum` (lines 18 and 34 of Algorithm 5).

Server side of dynamic-weighted atomic storage. Algorithm 6 is the pseudo-code of the servers' algorithm. The servers' algorithm is similar to the servers' algorithm of the ABD protocol with only one difference: each server includes its set of changes, $\mathcal{C}$, to its responses.

Correctness. We show that our dynamic-weighted atomic storage satisfies the safety and liveness properties of an atomic register according to the following definition:

Definition 6 (Atomic register [30])**.** Assume two read operations r_1 and r_2 executed by correct processes. Consider that r_1 terminates before r_2 initiates. If r_1 reads a value α from register R, then either r_2 reads α or r_2 reads a more up-to-date value than α.

Lemma 3. Assume there is no transfer from time t_1 to time t_2. Also, assume that set $S_1 \subseteq \mathcal{S}$ (resp. $S_2 \subseteq \mathcal{S}$) is determined as a quorum by function `is_quorum` such that every server $s_1 \in S_1$ (resp. every server $s_2 \in S_2$) is contacted from t_1 to t_2. Then, two quorums S_1 and S_2 have a non-empty intersection, i.e., $S_1 \cap S_2 \neq \emptyset$.

Proof. A set $S' \subseteq \mathcal{S}$ is determined as a quorum by function `is_quorum` if the total weight of servers in S' is greater than $\frac{\mathtt{W}_{\mathcal{S},0}}{2}$. For contradiction, assume that $S_1 \cap S_2 = \emptyset$. Sets S_1 and S_2 are determined as quorums. Let tw_1 and tw_2 be equal to the total weight of servers in S_1 and S_2, respectively. We have:

$$\begin{cases} \frac{\mathtt{W}_{\mathcal{S},0}}{2} < tw_1 \\ \frac{\mathtt{W}_{\mathcal{S},0}}{2} < tw_2 \end{cases} \Rightarrow \mathtt{W}_{\mathcal{S},0} < tw_1 + tw_2$$

Since $S_1 \cap S_2 = \emptyset$, the total weight of servers $\{S_1 \cup S_2\}$ is equal to $tw_1 + tw_2 > \mathtt{W}_{\mathcal{S},0}$; on the other hand, we know that $\{S_1 \cup S_2\} \subseteq \mathcal{S}$ and the total weight of servers $\mathcal{S}$ is equal to $\mathtt{W}_{\mathcal{S},0}$; hence, we find a contradiction. □

Lemma 4. Assume that a `read` operation r_1 returns $\langle tag_\alpha, \alpha\rangle$ at time t^e_α. Also, assume that another `read` operation r_2 started

Algorithm 5 The reader-writer side of the `read`/`write` protocols - process p_i.

variables
1: $opCnt \leftarrow 0$
2: $\mathcal{C} \leftarrow \{\langle s, 1, s, 1\rangle \mid \forall s \in \mathcal{S}\}$

functions
3: `read`() $\equiv$ `read_write`($\perp$)
4: `write`($value$) $\equiv$ `read_write`($value$)

function is_quorum(Q)
5: **if** $\frac{\mathtt{W}_{\mathcal{S},0}}{2} < sum\big(\{\mathtt{W}_{s_i,*} \mid s_i \in Q\}\big)$
6: **return** yes
7: **else**
8: **return** no

function `read_write`($value$)
`phase1`
9: $opCnt \leftarrow opCnt + 1$
10: **send** $\langle \mathtt{R}, opCnt\rangle$ to all servers
11: $Q \leftarrow \emptyset$
12: **repeat**
13: **upon receipt of** $\langle \mathtt{R\_A}, reg, opCnt, \mathcal{C}'\rangle$ from s_i
14: **if** $\mathcal{C} \neq \mathcal{C}'$
15: $\mathcal{C} \leftarrow \mathcal{C}'$
16: `read_write`($value$) ▷ restart the operation
17: $Q \leftarrow Q \cup s_i.\langle reg, \mathcal{C}\rangle$
18: **until** `is_quorum`(Q)
19: $maxtag \leftarrow \mathtt{max}\big(\{s_i.reg.tag \mid s_i \in Q\}\big)$
20: $maxreg \leftarrow \mathtt{find}\big(\{s_i.reg \mid s_i \in Q$ **and** $s_i.reg.tag = maxtag\}\big)$
21: **if** $value = \perp$
22: $value \leftarrow maxreg.value$
23: **else**
24: $ts \leftarrow maxtag.ts + 1$
25: $pid \leftarrow p_i$
`phase2`
26: **send** $\langle \mathtt{W}, \langle\langle ts, pid\rangle, value\rangle, opCnt\rangle$ to all servers
27: $Q \leftarrow \emptyset$
28: **repeat**
29: **upon receipt of** $\langle \mathtt{W\_A}, reg, opCnt, \mathcal{C}'\rangle$ from s_i
30: **if** $\mathcal{C} \neq \mathcal{C}'$
31: $\mathcal{C} \leftarrow \mathcal{C}'$
32: `read_write`($value$) ▷ restart the operation
33: $Q \leftarrow S \cup s_i.\langle reg, \mathcal{C}\rangle$
34: **until** `is_quorum`(Q)
35: **return** $value$

Algorithm 6 The server side of the `read`/`write` protocols - server s_i.

upon receipt of $\langle \mathtt{R}, cnt\rangle$ from p
1: **send** $\langle \mathtt{R\_A}, register, cnt, \mathcal{C}\rangle$ to p

upon receipt of $\langle \mathtt{W}, \langle tag, val\rangle, cnt\rangle$ from p
2: **if** $register.tag < tag$
3: $register \leftarrow \langle tag, val\rangle$
4: **send** $\langle \mathtt{W\_A}, cnt, \mathcal{C}\rangle$ to p

at time $t^s_\beta > t^e_\alpha$ returns $\langle tag_\beta, \beta\rangle$. If there is only one transfer at time t such that $t^e_\alpha < t < t^s_\beta$, one of the following cases happen: (1) $tag_\alpha = tag_\beta$ and $\alpha = \beta$, or (2) $tag_\alpha \leq tag_\beta$ and β was written after α.

Proof. Without loss of generality, assume that the transfer increases the weight of a server s_i and decreases the weight of a server s_j. Since there is only one transfer, the weights of other servers are not reassigned. Reassigning the weights of s_i and s_j is the only factor that causes the constitution of new quorums; since before accomplishing such a transfer, server s_i executes a `read` operation to update its register (Algorithm 4, lines 8-9), this lemma is similar to Lemma 3. □

Lemma 5. Assume that a `read` operation r_1 returns $\langle tag_\alpha, \alpha\rangle$ at time t^e_α. Also, assume that another `read` operation r_2 started at time $t^s_\beta > t^e_\alpha$ returns $\langle tag_\beta, \beta\rangle$. If there is at least one transfer at time t such that $t^e_\alpha < t < t^s_\beta$, one of the following cases happen: (1) $tag_\alpha = tag_\beta$ and $\alpha = \beta$, or (2) $tag_\alpha \leq tag_\beta$ and β was written after α.

Proof. This lemma is straightforward using Lemma 4. □

Lemma 6. Assume that a `read` operation r_1 returns a value α_1 at time t^e_1 with an associated tag tag_1. Also, assume that another `read` operation r_2 started at time $t^s_2 > t^e_1$ returns a value α_2 associated with tag tag_2. Then, one of the following cases happens: (1) $tag_1 = tag_2$ and $\alpha_1 = \alpha_2$, or (2) $tag_1 \leq tag_2$ and α_2 was written after α_1.

Proof. Assume that r_1 starts at time t^s_1 and r_2 ends at time t^e_2, then $t^s_1 < t^e_1 < t^s_2 < t^e_2$. There are four mutually exclusive cases:

a) There is no concurrent transfer. For this case, the lemma holds according to Lemma 3.
b) There is a concurrent transfer with r_1 at time t' such that $t^s_1 < t' < t^e_1$. For this case, the lemma holds according to Lemma 4.
c) There is at least one transfer between r_1 and r_2. For this case, the lemma holds according to Lemma 5.
d) There is a concurrent transfer with r_2 at time t' such that $t^s_2 < t' < t^e_2$. This case is similar to Case 2.

□

Proof of Theorem 6. Without loss of generality, we assume that this operation is invoked a finite number of times, like the asynchronous reconfiguration problem [31], [32]. According to Lemma 6, the storage system implemented using the `read`/`write` protocols (Algorithm 5 and Algorithm 6) is atomic storage. Since at most f servers might fail, there are $n - f$ servers in the worst case. The minimum value for the total weight of $n-f$ servers is greater than $\frac{\mathtt{W}_{\mathcal{S},0}}{2}$. Consequently, a quorum can be constituted, i.e., the system remains live even in the worst-case scenario.