Revisiting weighted quorums and asynchronous reconfiguration for atomic storage

Hasan Heydari Gharehbolagh

▶ **To cite this version:**

Hasan Heydari Gharehbolagh. Revisiting weighted quorums and asynchronous reconfiguration for atomic storage. Other [cs.OH]. Université Paul Sabatier - Toulouse III, 2022. English. NNT : 2022TOU30075 . tel-03726541

HAL Id: tel-03726541
https://theses.hal.science/tel-03726541

Submitted on 18 Jul 2022

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.

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par l'Université Toulouse 3 - Paul Sabatier

Présentée et soutenue par

Hasan HEYDARI GHAREHBOLAGH

Le 21 mars 2022

Revisiter les quorums pondérés et les reconfigurations asynchrones pour les systèmes de stockage atomique

---

Ecole doctorale : **SYSTEMES**

Spécialité : **Informatique et Systèmes Embarqués**

Unité de recherche :
ENAC-LAB - Laboratoire de Recherche ENAC

Thèse dirigée par
Alain PIROVANO et Guthemberg DA SILVA SILVESTRE

Jury

M. Arnaud CASTEIGTS, Rapporteur
M. Achour MOSTEFAOUI, Rapporteur
Mme Colette JOHNEN, Examinatrice
M. Franck MORVAN, Examinateur
M. Alain PIROVANO, Directeur de thèse
M. Guthemberg SILVESTRE, Co-directeur de thèse

Acknowledgements

I have received a great deal of support and assistance during my Ph.D.

First and foremost, I have to thank my supervisor, Professor *Guthemberg Silvestre*, whose expertise was invaluable in formulating the research questions and methodology. His insightful feedback pushed me to sharpen my thinking and brought my work to a higher level. Besides, his support for accomplishing internships and participating in scientific events aided me considerably.

My sincere thanks go to Professors *Luciana Arantes* and *Pierre Sens* at Sorbonne University and *Alysson Bessani* at Universidade de Lisboa, who allowed me to join their teams as an intern and gave me access to the laboratories and research facilities. Without their precious support, it would not be possible to conduct this research.

Besides, I thank the members of the Resco team at ENAC, especially Professor *Alain Pirovano*, for the stimulating discussions. They all have played a significant role in improving my thesis.

Last but not least, I would like to thank my family– my parents and wife– for their invaluable support. They gave me enough moral support, encouragement, and motivation to accomplish my personal goals. I consider myself nothing without them.

Contents

1 Introduction **13**
1.1 Motivation 15
1.2 Research Problems 17
1.3 Roadmap 19

2 Preliminaries and Background **21**
2.1 System Model 21
2.2 Definitions Related to Quorum Systems 21
2.3 Weight Assignment 22
2.4 Weight Reassignment 24
2.5 Atomic Storage 25
2.6 Nomenclature 26

3 Weight Reassignment Approaches **29**
3.1 Warmup 29
3.2 IncreaseDecrease Approach 30
3.3 PerfectPairwise Approach 33
3.4 WeakPairwise Approach 36
3.5 Discussion 38
3.5.1 Weight Reassignment vs. Reconfiguration 38
3.5.2 Weight Reassignment vs. Asset Transfer 40

4 Epoch-Based Weight Reassignment **43**
4.1 Warmup 43
4.2 Preliminaries 44
4.3 PairwiseWR Algorithm 46
4.4 EpochChanger Algorithm 47
4.5 Read-Write Protocols 52
4.6 Monitoring System 57
4.7 Performance Evaluation 59

5 Epoch-Less Weight Reassignment **63**
5.1 Warmup 63
5.2 Donate Operation 64

5.3 Retake Operation 67
5.4 Optimization 69
5.5 Read-Write Protocols 70

6 Conclusion **75**

List of Figures

1.1 The quorum latency of MQS vs. WMQS 15

2.1 The way of illustrating servers' latency scores 23

3.1 An example of the IncreaseDecrease weight reassignment abstraction 31
3.2 The illustration of the PerfectPairwise approach. Operation donate(sn, w, s_j) is called by server s_i, and two changes are returned to servers s_i and s_j. Consequently, the weight of server s_i (resp. s_j) decreases by w', where $w' = 0$ or w (resp. increases by w'', where $w'' = 0$ or w). 34
3.3 An example of losing weight in the WeakPairwise approach 38

4.1 Executions of the donate and retake operations 44
4.2 An example of executing executing the PairwiseWR and the EpochChanger algorithms. The weight of each server in epoch e_1 (resp. epoch e_2) is shown at the top of epoch e_1 (resp. epoch e_2). 50
4.3 The relationship between the clients and the monitoring system . 58
4.4 ClientServerMS module of the monitoring system 59
4.5 Quorum latency evaluation for our protocol, MW-ABD, and RAMBO. 61

5.1 Ilustration of donate and retake operations in the epoch-less weight reassignment protocol 64
5.2 An example of violating the safety property 67
5.3 An example of calling donate, retake, and clients' read-write operations 74

List of Tables

2.1 The symbols and their meanings used throughout the manuscript 26

Abstract

In the era of big data, cryptocurrencies, and the internet of everything, distributed storage systems are demanding more than ever. These systems use replication, which stores copies of data on multiple storage units called servers, to provide fault tolerance and high availability. Since each server might store a different version of data, ensuring consistency is an essential issue in distributed storage systems. A conventional technique to ensure consistency is using quorum systems. Among different types of quorum systems, the majority quorum is the most used one due to its simplicity and optimal fault tolerance.

A traditional approach to enhance the performance of distributed storage systems based on the majority quorum is to use the weighted quorum (a.k.a, weighted voting). In the weighted quorum, a static weight is assigned to each server based on its performance, and the quorums are constituted by considering the assigned weights. A significant limitation of the weighted quorum is its reliance on static weight assignments, which are inappropriate for dynamic environments and long-lived systems. To overcome such a limitation, the dynamic weighted quorum (a.k.a, dynamic weighted voting) is used. The dynamic weighted quorum utilizes weight reassignment protocols to reassign weights over time according to the performance variations detected by monitoring systems.

Many weight reassignment protocols have been proposed based on consensus or similar primitives. However, it is well-known that consensus cannot be implemented in asynchronous, failure-prone distributed systems. Hence, a distributed storage system based on the dynamic weighted quorum that uses a consensus-based weight reassignment protocol cannot be implemented in asynchronous and failure-prone distributed systems. On the other hand, some distributed storage systems, such as atomic storage, can be implemented in asynchronous and failure-prone distributed systems, for which an interesting problem is the following one: "is there any consensus-free weight reassignment protocol?"

No study has been dedicated to investigating the consensus-free weight reassignment protocols to the best of our knowledge. As one of the contributions of this thesis, we investigate three weight reassignment approaches determining whether they can be implemented without consensus in asynchronous, failure-prone distributed systems.

We first investigate the most intuitive approach for reassigning weights in which only the weight of one server can be decreased or increased while the weights of other servers remain unchanged in each call of the provided oper-

ations. In this approach, reassigning even one weight affects the total weight of servers. Accordingly, the total weight of servers should be recomputed by performing each weight reassignment. We treat such an approach as a concurrent object; then, we prove that its consensus number is greater than one to show that it cannot be implemented in asynchronous, failure-prone distributed systems, regardless of how it is implemented.

In the second approach, the weights are reassigned in a pairwise manner where each server can donate some part of its weight to another server. Besides, each server can retake its donated weights. The advantage of this approach against the first approach is that the total weight of servers does not change over time, so it is not required to recompute the total weight of servers by performing each weight reassignment. Similarly, we show that the second approach cannot be implemented in asynchronous, failure-prone distributed systems. Finally, we investigate another approach that is a weaker version of the second approach and has an asynchronous implementation.

Another contribution of this thesis is to construct atomic storage based on the dynamic weighted quorum in which weights can be reassigned without consensus over time using the implementation of the third weight reassignment approach. To this end, we first consider the crash failure model and present an epoch-based protocol. Then, to improve the efficiency of the epoch-based protocol, we present an epoch-less protocol.

A few problems related to the dynamic weighted quorum, weight reassignment approaches, and weight reassignment protocols are considered in this thesis as well. One of the related problems is determining essential criteria for weight assignments to avoid liveness and safety issues. Designing an efficient monitoring system to determine the performance variations is another problem. The reconfiguration problem in which the set of servers changes over time is investigated due to its similarity to the weight reassignment problem, and a new impossibility result is presented.

Chapter 1

Introduction

In the era of big data, cloud computing, cryptocurrencies, and the internet of everything, storage systems are demanding more than ever. Since such systems are prone to various types of failures, like disk failures and failures created by malicious attacks, they should be fault-tolerant. Replication is the most well-known technique to construct fault-tolerant storage systems [23, 41, 57, 65]. In a replicated storage system, copies of data are stored in multiple storage units, so the system remains available, i.e., respond to clients in a timely manner [50], by occurring failures for a proper subset of its storage units.

A replicated storage system can be located in a single physical site and connected to the clients via a single network interface, like RAID systems [53]. Such a storage system constitutes a single point of failure so that data cannot survive after complete site disasters. In contrast, storage units can be placed in several sites constructing a distributed storage system. Each storage unit in a distributed storage system is a cheap commodity disk or low-end PC [14]. These distinguishing features and many others cause distributed storage systems to become increasingly popular.

In a distributed storage system, storage units are called servers. Each copy of data stored by a server might have a different version than other copies, so ensuring consistency is a significant problem. Intuitively, *consistency* means that every client should be able to execute the provided operations, like read and write, on the most up-to-date version of data stored by the storage system [12, 41]. However, providing both consistency and availability in distributed storage systems is impossible in the presence of network partitions due to the CAP theorem [24].

There are two main approaches to designing a distributed storage system based on that impossibility result. The first approach considers network partitions and presents a trade-off between consistency and availability, like NoSQL distributed databases [12]. The other approach excludes network partitions and provides both consistency and availability by assuming that at least one available quorum exists to execute each operation [41]. In further detail, servers are organized as a quorum system [66], and each operation should be executed by a

quorum. A quorum system is a collection of sets called quorums such that each quorum is a subset of servers, and the *intersection* property that states every two quorums intersect should be satisfied. The *intersection* property guarantees that no operation can miss the most up-to-date version of data.

There exist many types of quorum systems such as grids [13, 49], trees [3], hierarchical [40], and the majority quorum system (MQS) [67]. In the MQS, also known as majority voting [62, 64], every quorum consists of a strict majority of servers. Most replicated storage systems based on quorum systems, such as ZooKeeper [35] and Etcd [2], utilize the MQS due to its simplicity and optimal fault tolerance; however, the MQS can impact both quorum latency[1] and throughput [67]. The reason for this performance impact is that an MQS does not consider the heterogeneity of servers or network connections. If it takes such heterogeneity into account, its quorum latency and throughput are likely to be improved.

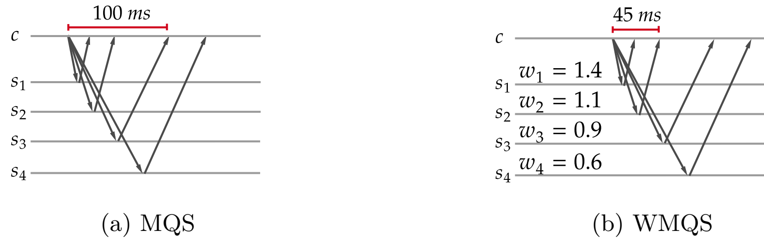
Contrarily to MQS, the weighted majority quorum system (WMQS), also known as weighted voting [7, 39, 52] or weighted replication [11], was proposed to cope with heterogeneity. In the WMQS, each server is assigned a weight or voting power that is in accordance with the server's latency or throughput determined by a monitoring system [10, 11]; every quorum consists of a set of servers such that the sum of their weights is greater than half of the total weight of servers in the system. This way, proportionally smaller quorums can be constituted so the performance of the system can be improved. The following example helps to grasp the difference between MQS and WMQS in distributed storage systems with heterogeneous latencies and throughput.

Example 1. Let s_1, s_2, s_3, and s_4 be the servers comprising the distributed storage 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 servers s_1, s_2, s_3, and s_4 are $20ms$, $45ms$, $100ms$, and $140ms$, respectively. In another scenario, assume that the throughput of servers s_1, s_2, s_3, and s_4 are 1000, 800, 400, and 200 operation/sec, respectively. Such latencies and throughput are determined using a monitoring system. Let 1.4, 1.1, 0.9, 0.6 be the assigned weights to servers s_1, s_2, s_3, and s_4, respectively. The quorum latency using MQS is $100ms$ while using WMQS is $45ms$ (Figure 1.1). The throughput of the system based MQS and WMQS is 600 and 800 operation/sec, respectively[2]. Both scenarios show the advantage of using the WMQS over MQS due to constituting proportionally smaller quorums.

The WMQS relies on static weight assignments, so it has a significant limitation for dynamic environments and long-lived distributed storage systems, where the latencies and throughput of servers might change over time. To overcome such a limitation, the dynamic weighted majority quorum system (DWMQS), also known as dynamic voting [15, 36, 68], can be used. The DWMQS utilizes

[1]Quorum 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 servers) until receiving the responses from a quorum of servers [51, 67].

[2]Throughput is computed using *quoracle* library [67].



(a) MQS (b) WMQS

Figure 1.1: The quorum latency of MQS vs. WMQS

weight reassignment protocols to reassign weights over time according to the performance variations detected by monitoring systems.

Many weight reassignment protocols have been proposed based on consensus[3] or similar primitives (for example, the protocols presented in [11, 15, 36]). The general scheme of these protocols is as follows: new weights are proposed; then, servers execute a consensus protocol (for example, Paxos [43]) or similar primitives to reach an agreement on the proposed weights; finally, the decided weights are used. The objective of using consensus or similar primitives is to ensure that all processes use the same set of weights to decide whether a subset of servers constitutes a quorum. An important feature of these protocols is that they cannot be implemented in asynchronous and failure-prone distributed systems due to the FLP theorem [19].

Using the weight reassignment protocols based on consensus or similar primitives for consensus-based distributed storage systems, like state machine replication [56], is not problematic. However, some distributed storage systems can be implemented in asynchronous, failure-prone distributed systems for which consensus-free weight reassignment protocols should be used to avoid changing the timing model. Regarding consensus-free weight reassignment protocols, no study has been done to the best of our knowledge, and studying such protocols is the primary objective of this thesis.

1.1 Motivation

One of the fundamental distributed storage systems is the atomic storage system (simply atomic storage, also known as atomic register [42]). Although such a storage system has only two operations– read and write– it provides the building blocks for both complex practical storage systems (for example, FAB [54] and GPFS [55]) and theoretical storage systems (like, lattice agreement [9]). Atomic storage ensures consistency as follows: each read returns the *last* value written.

[3] Consensus is one the most fundamental problems of distributed systems [16, 29, 63]. In case a set of servers requires to agree on a value, consensus is used. Any solution for consensus must satisfy three properties: *agreement*: no two servers decide on different values; *validity*: the decided value was proposed by some server; *termination*: all correct servers reach a decision. Although consensus is a powerful primitive in designing distributed services, it cannot be implemented in asynchronous, failure-prone distributed systems, even in the presence of a single crash failure [19].

An important feature of atomic storage is that it can be implemented in failure-prone asynchronous distributed systems [33, 66].

Regarding the number of readers and writers, there are two types of atomic storage [33]. The first type is the single-writer multi-reader (SWMR), in which there is only one process that is allowed to execute write operations while there are multiple processes that can execute read operations. The ABD protocol [8] is the first protocol presented to implement the SWMR atomic storage in crash-prone asynchronous distributed systems. Each read operation takes two communication rounds[4], where each communication round contains two phases[5]. On the other hand, each write operation takes one communication round that contains two phases. Several protocols have been presented to improve the efficiency of the ABD protocol by reducing the number of communication rounds or phases of the read operations (for example, the protocols presented in [17, 30, 47] and references therein).

Contrarily to the SWMR atomic storage, there is another type called multi-writer multi-reader (MWMR) atomic storage, in which there are multiple processes that can execute write operations. Lynch and Schwarzmann presented the first MWMR extension of the ABD protocol in [45]. Each read or write operation takes two communication rounds, where each communication round contains two phases. We name that protocol the MW-ABD. Several protocols have been presented to improve the efficiency of the MW-ABD protocol by decreasing the number of its communication rounds and phases (for example, [18, 22]).

The ABD, MW-ABD, and similar protocols are based on static configurations, where the set of servers does not change over time. Although atomic storage is able to tolerate failures of some servers, servers might require to be replaced in long-lived systems. Moreover, some new servers might be added to the system to enhance the performance. Therefore, the protocols of atomic storage must be able to provide reconfiguration operations by which new servers can be added, and old servers can be removed [4, 60].

Multiple consensus-based protocols for reconfigurable atomic storage have been proposed (like, [21, 25, 27]). However, researchers have tried to propose consensus-free protocols for reconfigurable atomic storage due to the possibility of implementing atomic storage in asynchronous, failure-prone distributed systems. Aguilera et al. proposed the first consensus-free protocol for reconfigurable atomic storage called DynaStore [5]. In this protocol, atomic storage uses a default configuration that is known by all servers initially. The protocol provides three kinds of operations: read, write, and reconfig. The read and write operations are executed similar to the MW-ABD protocol. Servers can change their current configurations using reconfig operation. The drawback of DynaStore is its performance. Several protocols (for example, [6, 37, 38, 61]) have

[4]A process p performs a communication round during an operation if (1) process p sends a message for the operation to a subset of servers, (2) upon delivery of the message, each server sends its response to process p, and (3) when process p receives a sufficient number of responses from servers, it terminates the communication round [31].

[5]Within an execution of a communication round, a phase is the set of sends and corresponding matching received responses [31].

been proposed to improve the performance of DynaStore.

To enhance the performance of atomic storage based on the MQS, instead of decreasing the number of communication rounds or phases, one can replace the MQS with WMQS. Since the servers' performance might change over time in long-lived systems, the weights should be reassigned over time; therefore, using the DWMQS instead of the WMQS is a better choice. At this point, we can ask a similar question to the reconfigurable atomic storage: "can we construct a consensus-free atomic storage based on DWMQS?"

1.2 Research Problems

The research done in this thesis can be summarized as addressing three main problems. First, we investigate the weight reassignment problem by considering three weight reassignment approaches to determine whether they can be implemented without consensus (Problem 1). Second, we design two consensus-free protocols for the weight reassignment problem. Then, we construct consensus-free atomic storage based on DWMQS such that the servers' weights are reassigned using the proposed consensus-free weight reassignment protocols (Problem 2). Third, we investigate the relationship between two problems– the weight reassignment and reconfiguration problems– and present a new variation for the reconfiguration problem (Problem 3). Each problem is described in further detail below.

Problem 1: investigating the consensus-free weight reassignment problem

To investigate the consensus-free weight reassignment problem, we take two steps. We first define the problem of weight reassignment for the DWMQS. For a system based on the MQS, every quorum consists of a strict majority of servers; therefore, a majority of servers should be correct to guarantee the system's availability. In other words, the maximum number of failed servers, f, should be less than equal to $\lfloor (n-1)/2 \rfloor$, where n denotes the number of servers. In case the MQS is replaced with the WMQS, to guarantee the system's availability, the weights should be assigned so that the total of the f greatest weights is less than $tw/2$, where tw denotes the total weight of servers. Similarly, if the WMQS is replaced with the DWMQS, the total of the f greatest weights should remain less than $tw/2$ over time.

Based on the relationship between the f greatest weights and tw, we formulate the fundamental problem of the DWMQS and weight reassignment protocols as follows:

> "Assume that a system based on DWMQS is available initially, i.e., the total of the f greatest weights is less than $tw/2$. Multiple concurrent requests are issued to reassign weights. Consequently, the total weight of servers might change after applying a subset of such requests. Is there any weight

> reassignment protocol that satisfies the following properties: (1) it can be implemented in asynchronous distributed systems and (2) it guarantees that the total of the f greatest weights remains less than half of the total weight of servers?"

Presenting a protocol that only satisfies property (2) is straightforward using consensus or similar primitives. However, satisfying both properties is a challenging problem.

For the second step, we consider three weight reassignment approaches. A weight reassignment approach indicates how weight reassignment requests can be issued and processed. We investigate whether these approaches can be used in designing protocols for the fundamental problem of the DWMQS and weight reassignment protocols. To this end, we use a notion called *consensus number* introduced by Herlihy in [32]. Each type T has an associated *consensus number*, which is the maximal number of servers that can solve consensus using objects of type T and atomic registers [28, 32]. For instance, the consensus number of queue is two [32], which means that two servers can solve consensus using objects of queue and atomic registers while three servers cannot. Recently, Guerraoui et al. used this notion to show that cryptocurrencies can be implemented without consensus [28].

We employ consensus numbers according to the following reasoning throughout this thesis: (a) we know that consensus cannot be implemented in asynchronous, failure-prone distributed systems [19]; (b) we show that consensus can be solved among n servers using an object of type T and some message-passing primitives that can be implemented in asynchronous, failure-prone distributed systems (i.e., accordingly, the consensus number of type T is greater than equal to n); from (a) and (b), we conclude that an object of type T cannot be implemented in asynchronous, failure-prone distributed systems.

We first consider the most intuitive approach for reassigning weights in which only the weight of one server can be decreased or increased while the weights of other servers remain unchanged in each call of the provided operations. We call such an approach the IncreaseDecrease approach and treat it as a concurrent object; then, we prove that its consensus number is greater than one to show that it cannot be implemented in asynchronous distributed systems.

In the second approach called the PerfectPairwise, the weights are reassigned in a pairwise manner where each server can donate some part of its weight to another server. The PerfectPairwise approach is closely similar to the asset transfer problem [48]. In the asset transfer problem, each server has an account and can transfer some part of its balance to another server if the transferred balance does not turn its balance negative. Although it has been proved that the asset transfer problem has an asynchronous implementation [28], we show that the PerfectPairwise approach does not have an asynchronous implementation, similar to the IncreaseDecrease approach. Finally, we investigate a weaker version of the second approach called the WeakPairwise; and we show that it can be implemented in asynchronous failure-prone distributed systems.

Problem 2: designing protocols for the weight reassignment problem

We present two implementations for the WeakPairwise approach. We first propose an epoch-based implementation of the WeakPairwise approach. This implementation is similar to the view-based reconfiguration protocols, like FreeStore [6]. Besides, we design an efficient monitoring system for atomic storage. We present the read-write protocols of atomic storage that are based on the DWMQS, and the weights are reassigned using the proposed epoch-based implementation of the WeakPairwise approach. Next, to improve the performance of the epoch-based protocol, we propose another protocol to implement the WeakPairwise approach. This protocol does not rely on epochs in contrast to the epoch-based protocol.

Problem 3: investigating the relationship between the weight reassignment and reconfiguration problems

The reconfiguration problem is the problem of changing the set of servers over time, i.e., some servers can be removed from a system, or some new servers can be joined to the system. Multiple consensus-free protocols for implementing the reconfigurable atomic storage have been proposed [5, 6, 37, 38, 61]. On the other hand, it has been proved that it is impossible to reconfigure a storage system infinitely many times while guaranteeing the liveness of the storage system [59].

In both reconfiguration and weight reassignment problems, quorum systems change over time. Based on this relationship, we introduce a variant of the asynchronous reconfiguration problem called the *conditional reconfiguration* problem. The *conditional reconfiguration* problem is similar to the standard asynchronous reconfiguration problem with only one difference: a condition related to the size of configurations should be satisfied. Similar to the impossibility result presented for the IncreaseDecrease approach, we show that the *conditional reconfiguration* problem has no asynchronous algorithm.

1.3 Roadmap

The remainder of this thesis is organized into six chapters.

- In Chapter 2, we provide preliminaries and background for our contributions. We present a detailed system model. Then, we provide some fundamental assumptions and properties for weight assignment and weight reassignment problems. Then, we define the problem of atomic storage.

- In Chapter 3, we investigate the problem of weight reassignment by considering three weight reassignment approaches– IncreaseDecrease, PerfectPairwise, and WeakPairwise. We use several examples to highlight the properties of approaches. Then, the impossibility results of the IncreaseDecrease and PerfectPairwise approaches are presented. Furthermore, the reconfiguration and

asset transfer problems and their relationships with the weight reassignment problem are discussed.

- In Chapter 4, we propose an epoch-based implementation of the WeakPairwise approach. Then, we design an efficient monitoring system for atomic storage. We present the read-write protocols of atomic storage that are based on the DWMQS, and the weights are reassigned using the proposed epoch-based implementation of the WeakPairwise approach.

- In Chapter 5, we propose another protocol to implement the WeakPairwise approach. This protocol does not rely on epochs in contrast to the protocol presented in Chapter 4. We discuss why an epoch-less protocol is more efficient than its epoch-based counterpart.

- Finally, we present the conclusion and provide possible future directions for this work in Chapter 6.

Chapter 2

Preliminaries and Background

Abstract. In this chapter, we provide preliminaries and background for our contributions. We present a detailed system model. We also present a few definitions related to quorum systems. Then, we provide some fundamental assumptions, properties, and definitions for weight assignment and weight reassignment problems. Finally, we present the problem of atomic storage.

2.1 System Model

We consider a distributed system composed of 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. The processes communicate by message passing, and the links reliably connect all pairs of processes. 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 system's lifetime. We consider the crash fault-tolerant (CFT) model throughout this manuscript. In this model, processes are prone to crash failures, and a process is called *correct* if it is not crashed. We assume that at most f servers might crash.

2.2 Definitions Related to Quorum Systems

Our work is based on quorum systems, so we present a few definitions related to quorum systems in this section. We begin by presenting the definition of quorum systems.

Definition 1 (quorum system [66])**.** A quorum system is a collection of sets called quorums such that each quorum is a subset of servers, and the *intersection*

property that states every two quorums intersect should be satisfied.

There exist many types of quorum systems; among them, the majority quorum system (MQS) has significant importance due to its simplicity and optimal fault tolerance [67]. The definition of the MQS is as follows.

Definition 2 (majority quorum system (MQS) [67])**.** The MQS is the one that every quorum consists of a strict majority of servers.

The weighed majority quorum system (WMQS) can be used to enhance the performance of the systems based on the MQS. In the WMQS, each server is assigned a weight or voting power that is in accordance with the server's latency or throughput determined by a monitoring system [10, 11]. Then, the weights are used to determine whether a subset of servers constitutes a quorum. The following definition is the definition of the WMQS.

Definition 3 (weighted majority quorum system (WMQS))**.** The WMQS is the one that every quorum consists of a set of servers whose total weight is greater than half of the total weight of all servers.

2.3 Weight Assignment

In the WMQS, a weight should be assigned to each server. One can assign a weight to each server based on different criteria, such as throughput or latency of the server. Note that throughput can be effectively improved by adding more resources (CPU, memory, faster disks) to servers or using better links. However, latency in geo-distributed storage systems will be affected by the speed of light limit and perturbations caused by bandwidth sharing [58]. Therefore, we focus on considering latency instead of throughput.

We assume that there is a monitoring system that assigns a score, called *latency score*, to each server based on the server's latency. The monitoring system provides a function called lscore that takes a server as input and returns the server's latency score. Based on the latency scores, the weigh reassignment protocols can assign weights to servers such that if the latency score of server s_i is greater than s_j, then $w_i < w_j$. For example, assume that the latency scores of servers s_1, s_2, s_3, and s_4 are 20, 45, 100, and 140, respectively. By assigning 1.4, 1.1, 0.9, 0.6 as the weights of servers s_1, s_2, s_3, and s_4, respectively, the relationship between the servers' weights and latency scores is satisfied.

If the servers' weights are assigned based on the servers' latency scores, as explained above, the quorum latency (see definition below) of the storage system will be enhanced on average. Chapter 4 presents further details about the monitoring system and the mentioned claim about enhancing the quorum latency.

Definition 4. Quorum 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 servers) until receiving the responses from a quorum of servers [51, 67].

Some figures presented throughout this manuscript use gray areas to show the servers' latency scores. The higher the height of a gray area associated with a server, the lower the server's latency score. For instance, the latency score of server s_1 is lower than server s_2 in Figure 2.1.

Figure 2.1: The way of illustrating servers' latency scores

Throughout this manuscript, the initial weight of servers, the weight of each server s_i, and the total weight of servers are denoted by iw, w_i, and tw, respectively. According to Definition 3, the total weight of any constituted quorum should be greater than $tw/2$ to satisfy the *intersection* property of the WMQS.

To guarantee the availability of a storage system based on the WMQS, a relationship between the servers' weights and f should be satisfied. The following property determines such a relationship.

Property 1 The summation of f greatest weights should be less than $tw/2$.

Let $f = \lfloor (n-1)/2 \rfloor$, like the MQS. Accordingly, the size of any quorum is $\lceil (n+1)/2 \rceil$ in the WMQS, regardless of the way of assigning weights to the servers, so there is no difference between the sizes of quorums in the MQS and WMQS. Recall from Example 1 presented in Chapter 1 that if proportionally smaller quorums are constituted the performance of the system can be enhanced. Therefore, it is required to assume that $f \leq \lfloor (n-1)/2 \rfloor$ to have proportionally smaller quorums in the WMQS. The following assumption states such an assumption by introducing a new parameter denoted by Δ.

Assumption 1 The relationship between n and f is as follows: $n = 2f + 1 + \Delta$, where Δ is a natural number.

By using Assumption 1 when $\Delta \geq 1$, the fault-tolerance of the system might be decreased while its performance can be improved due to having proportionally smaller quorums. A similar assumption is used by other weight assignment protocols (e.g., [58]). Weights can be determined in different ways. In the following, we present a weight reassignment scheme by which the weights can be determined.

WHEAT scheme

WHEAT (WeigHt-Enabled Active replicaTion) [58] is a scheme to assign weights to the servers such that the weight of each server is either $minw$ or $maxw$. The following assumption determines the values of $minw$ and $maxw$.

Assumption 2 The values of $minw$ and $maxw$ are 1 and $1 + \Delta/f$, respectively.

Servers constitute two types of quorums using WHEAT. The first type is the quorums with size $f+1$ that has the least quorum size. The second type is the quorums with size $n-f$. The following assumption determines the total weight of servers, that can be computed based on the values of $maxw$ and f. Note that knowing the total weight of servers is necessary to determine whether a subset of servers constitutes a quorum based on Definition 2.

Assumption 3 The total weight of servers, tw, is equal to $2 \times maxw \times f + 1$.

2.4 Weight Reassignment

In the DWMQS, the weight of each server might be reassigned over time. We use a data structure called *change* to store each weight reassignment of any server. A *change* is a triple $\langle w, s_i, \text{info} \rangle$, where $w \in \mathbb{R}$ is the amount of change made on the weight of server $s_i \in S$, and 'info' indicates the additional information about the *change*. Each server s_i has a local set denoted by *changes* to store every change $\langle *, s_i, * \rangle$. By using set *changes*, server s_i's weight can be computed as follows: $w_i = \mathsf{sum}\big(\{w \mid \forall \langle w, s_i, * \rangle \in s_i.changes\}\big)$. For example, let $s_i.changes = \big\{\langle 1, s_i, \text{init} \rangle, \langle 0.3, s_i, \text{increment} \rangle, \langle -0.1, s_i, \text{decrement} \rangle\big\}$. Then, server s_i's weight is equal to 1.2.

Based on changing or not changing the total weight of servers over time, there are two ways to reassign servers' weights:

- Time-varying total weight (TVTW), in which the total weight of servers can change over time, and

- Time-invariant total weight (TITW) in which the total weight of servers is time-invariant and known by every process in advance.

Note that to indicate that a subset of servers constitutes a quorum in the DWMQS, the total weight of servers should be known. Accordingly, using the TITW is simpler than the TVTW because it is not required to execute additional queries to determine the total weight of servers.

A fundamental problem related to the DWMQS and weight reassignment protocols that is considered in the following chapters is the following:

> "Assume that a system based on DWMQS is available initially, i.e., the total of the f greatest weights is less than $tw/2$. Multiple concurrent requests are issued to reassign weights. Consequently, the total weight of servers might change after applying a subset of such requests. Is there any weight reassignment protocol that satisfies the following properties: (1) it can be implemented in asynchronous distributed systems and (2) it guarantees that the total of the f greatest weights remains less than half of the total weight of servers?"

2.5 Atomic Storage

This section presents the basics of emulating a (static) distributed multi-writer multi-reader (MWMR) atomic storage based on quorum systems. Such a storage system allows processes to read, write, and overwrite a single value using the following operations:

– read() that returns the last written value or $\perp$ if no value was previously written, and

– write(val) that writes val.

The formal definition of atomic storage is as follows.

Definition 5 (Atomic storage [44]). An implementation of an object is atomic, if for any execution all the read and write operations that are invoked on an object complete, then the read and write operations for the object can be partially ordered by an ordering $\prec$, so that the following conditions are satisfied:

A1. There do not exist read or write operations π_1 and π_2 such that π_1 completes before π_2 starts, yet $\pi_2 \prec \pi_1$.

A2. All write operations are totally ordered and every read operation is ordered with respect to all the writes.

A3. Every read operation ordered after any write returns the value of the last write preceding it in the partial order; any read operation ordered before all writes return the initial value, $\perp$.

In the following, we present a summary of a protocol, the MW-ABD [45], that implements the MWMR atomic storage.

The MW-ABD protocol

The MW-ABD protocol [45] emulates a static distributed multi-writer multi-reader (MWMR) atomic storage based on quorum systems. The protocol consists of two different algorithms– reader-writer side[1] and server-side. The reader-writer side can be executed by a process (either a client or server) while the server-side is executed by servers.

Reader-writer side. Algorithm 1 is the pseudo-code of the MW-ABD's reader-writer side. Both read and write operations proceed in two phases, each ending with a confirmation that at least one quorum was accessed. In the first phase of a write operation (phase1), a writer contacts a quorum in order to determine the highest tag[2] $maxtag$ used prior to write the last written value (Lines 2-9 of

[1] In this manuscript, both clients and servers can execute read operations, so we use 'reader-writer side' instead of 'client-side.'

[2] A tag is a tuple $\langle ts, pid \rangle$, where ts and pid are a timestamp and the identifier of a process, respectively. Tag tag_1 is less than equal to tag tag_2 iff: ($tag_1.ts \leq tag_2.ts$) or ($tag_1.ts = tag_2.ts$ and $tag_1.pid \leq tag_2.pid$).

Algorithm 1). Then, in the second phase (phase2), the writer takes the following steps: (a) increments the timestamp of *maxtag*, (b) overwrites its identifier on the identifier of *maxtag*, (b) propagates its value, along with the changed *maxtag* within a WRITE message to a quorum (Lines 10-15 of Algorithm 1).

Similarly, in phase1 of a read operation, a reader contacts a quorum to retrieve the value associated with the highest tag *maxtag* reported by the quorum (Lines 16-24 of Algorithm 1). Then, in phase2, the reader propagates the read value along with *maxtag* within a WRITE message to a quorum (Lines 25-31 of Algorithm 1).

Server-side. Algorithm 2 is the pseudo-code of the MW-ABD's server-side. Each server s_i maintains a local variable called *register* used to store the tag and value of its local register. To access the tag (resp. value) of the local register, *register.tag* (resp. *register.val*) can be used. Each server updates its register locally when it receives a WRITE message with a higher tag than its own tag. Note that the WRITE message might be received from a writer or a reader executing its second phase.

2.6 Nomenclature

Table 2.1 indicates the nomenclature used throughout this manuscript.

Symbol	Meaning
$\mathcal{T}$	the lifetime of the system
Π	the set of all processes, either clients or servers
C	the set of clients
S	the set of servers
n	the total number of servers
m	the total number of clients
f	the maximum number of servers that can fail
tw	the total weight of servers
iw	the initial weight of each server
$minw$	the lower bound defined for servers' weights
$maxw$	the upper bound defined for servers' weights
$mrtl$	the maximum value for the round-trip latency of client-server communications

Table 2.1: The symbols and their meanings used throughout the manuscript

Algorithm 1 The reader-writer side of the MW-ABD - process p_i

variables
1) $opCnt \leftarrow 0$

function write($value$)
phase1
2) $opCnt \leftarrow opCnt + 1$
3) **send** $\langle \text{READ}, opCnt \rangle$ to all servers
4) $msgs \leftarrow \varnothing$
5) **repeat**
6) **upon receipt of** $\langle \text{READACK}, \langle tag, value \rangle, opCnt \rangle$ from s_i
7) $msgs \leftarrow msgs \cup \langle s_i, tag, value \rangle$
8) **until** servers in set $msgs$ constitutes a quorum
9) $maxtag \leftarrow \mathsf{max}\big(\{tag \mid \langle s_i, tag, value \rangle \in msgs\}\big)$
phase2
10) **send** $\langle \text{WRITE}, \langle \langle maxtag.ts + 1, p_i \rangle, value \rangle, opCnt \rangle$ to all servers
11) $msgs \leftarrow \varnothing$
12) **repeat**
13) **upon receipt of** $\langle \text{WRITEACK}, reg, opCnt \rangle$ from s_i
14) $msgs \leftarrow msgs \langle s_i, tag, value \rangle$
15) **until** servers in set $msgs$ constitutes a quorum

function read()
phase1
16) $opCnt \leftarrow opCnt + 1$
17) **send** $\langle \text{READ}, opCnt \rangle$ to all servers
18) $msgs \leftarrow \varnothing$
19) **repeat**
20) **upon receipt of** $\langle \text{READACK}, \langle tag, value \rangle, opCnt \rangle$ from s_i
21) $msgs \leftarrow msgs \cup \langle s_i, tag, value \rangle$
22) **until** servers in set $msgs$ constitutes a quorum
23) $maxtag \leftarrow \mathsf{max}\big(\{tag \mid \langle s_i, tag, value \rangle \in msgs\}\big)$
24) $maxreg \leftarrow \mathsf{find}(\langle s_i, tag, value \rangle \in msgs$ such that $tag = maxtag)$
phase2
25) **send** $\langle \text{WRITE}, \langle maxtag, maxreg.value \rangle, opCnt \rangle$ to all servers
26) $msgs \leftarrow \varnothing$
27) **repeat**
28) **upon receipt of** $\langle \text{WRITEACK}, reg, opCnt \rangle$ from s_i
29) $msgs \leftarrow msgs \langle s_i, tag, value \rangle$
30) **until** servers in set $msgs$ constitutes a quorum
31) **return** $value$

Algorithm 2 The server-side of the MW-ABD - server s_i

variables

1) $register[tag, val] \leftarrow \langle \varnothing, \perp \rangle$

upon receipt of $\langle \text{READ}, cnt \rangle$ from p

2) **send** $\langle \text{READACK}, register, cnt \rangle$ to p

upon receipt of $\langle \text{WRITE}, \langle tag, val \rangle, cnt \rangle$ from p

3) **if** $register.tag < tag$
4) $\quad register \leftarrow \langle tag, val \rangle$
5) **send** $\langle \text{WRITEACK}, cnt \rangle$ to p

Chapter 3

Weight Reassignment Approaches

Abstract. Using the weighted majority quorum system enhances the performance of distributed storage systems by allowing smaller quorums to constitute in contrast to using the majority quorum system. A significant limitation of the weighted majority quorum system is its reliance on static weights, which are inappropriate for dynamic environments and long-lived systems. To overcome such a limitation, the dynamic weighted majority quorum system, that utilizes weight reassignment protocols to reassign weights over time according to the performance variations detected by monitoring systems, can be used. Weight reassignment protocols can be based on different approaches. This chapter presents the abstract forms of three approaches to reassign servers' weights and investigates whether they can be implemented in asynchronous, failure-prone distributed systems. The weight reassignment, reconfiguration, and asset transfer problems are closely related; this chapter discusses their relationship as well.

3.1 Warmup

A conventual approach to enhance the performance of systems based on the majority quorum system (MQS) is to replace the MQS with the weighted majority quorum system (WMQS) [11, 34, 39, 58]. In the WMQS, a static weight is assigned to each server based on the server's performance detected by a monitoring system, and every subset of servers whose total weight is greater than $tw/2$ is a quorum, where tw is the total weight of all servers. Accordingly, to guarantee the availability of the system that is based on the WMQS, the weights should be assigned in such a way that the total of the f greatest weights is less than $tw/2$.

A significant limitation of the WMQS is its reliance on static weights, which are inappropriate for dynamic environments and long-lived systems [11]. To overcome such a limitation, the dynamic weighted majority quorum system

(DWMQS) can be used. The DWMQS utilizes weight reassignment protocols to reassign weights over time according to the performance variations detected by monitoring systems. In this chapter, we investigate the fundamental problem related to the DWMQS and weight reassignment protocols that states:

> "Assume that a system based on DWMQS is available initially, i.e., the total of the f greatest weights is less than $tw/2$. Multiple concurrent requests are issued to reassign weights. Consequently, the total weight of servers might change after applying a subset of such requests. Is there any weight reassignment protocol that satisfies following properties: (1) it can be implemented in asynchronous distributed systems, and (2) the total of the f greatest weights remains less than $tw/2$?"

To this end, we investigate three weight reassignment approaches– IncreaseDecrease, PerfectPairwise, and WeakPairwise. We treat the first two approaches as concurrent objects; then, we prove that their consensus numbers are greater than one, i.e., such approaches cannot be implemented in asynchronous, failure-prone distributed systems. However, the last approach, a weaker version of the second approach, can be implemented in asynchronous, failure-prone distributed systems. Such an investigation is essential because it indicates that if a weight reassignment protocol wants to be implemented in asynchronous failure-prone distributed systems, it does not have the freedom to choose any weight reassignment approach.

The IncreaseDecrease approach is closely related to the reconfiguration problem so that its impossibility result can be presented for the reconfiguration problem as well. Notably, we prove that it is impossible to present a reconfiguration protocol in asynchronous, failure-prone distributed systems if a condition related to the number of servers should be satisfied at any time. On the other hand, the PerfectPairwise approach is similar to the asset transfer problem. Since the asset transfer problem can be implemented in asynchronous, failure-prone distributed systems, such a discussion highlights why the PerfectPairwise approach cannot be implemented in asynchronous, failure-prone distributed systems.

3.2 IncreaseDecrease Approach

IncreaseDecrease is a TVTW-based weight reassignment approach in which only the weight of one server can be increased or decreased while the weights of other servers remain unchanged in each call of the operations provided for weight increment or decrement. Each server has a local and monotonically increasing sequence number sn. This approach provides the three following operations:

- increase(sn, w) that can be called by servers. Each server calls this operation to increase its weight by w.
- decrease(sn, w) that can be called by servers. Each server calls this operation to decrease its weight by w.

– $\mathsf{collect}()$ that can be called by processes and returns the weights of servers.

Such a sequence number is increased before calling both $\mathsf{increase}$ and $\mathsf{decrease}$ operations. Operation $\mathsf{increase}(sn, w)$ called by server s_i returns a change $\langle w', s_i, \langle \text{INC}, s_i, sn, w \rangle\rangle$ to s_i, where w' can be either equal to 0 or w. Besides, operation $\mathsf{decrease}(sn, w)$ called by server s_i returns a change $\langle w', s_i, \langle \text{DEC}, s_i, sn, w \rangle\rangle$ to s_i, where w' can be either equal to 0 or $-w$. Each returned change is added to set *changes*. This approach has the following properties:

– (IncreaseDecrease Validity) Before receiving a change $\langle *, s_i, \langle *, s_i, sn, w \rangle\rangle$, server s_i had called an operation $op(sn, w)$, where $op \in \{\mathsf{increase}, \mathsf{decrease}\}$.

– (IncreaseDecrease Termination) Given operation $op(sn, w)$ called by server s_i such that $op \in \{\mathsf{increase}, \mathsf{decrease}\}$, server s_i eventually receives a change $\langle *, s_i, \langle *, s_i, sn, w \rangle\rangle$ if s_i is a correct server.

– (IncreaseDecrease Accuracy) Given operation $op(sn, w)$ called by server s_i such that $op \in \{\mathsf{increase}, \mathsf{decrease}\}$, one of the following cases can happen:

  – A change $\langle w' = 0, s_i, \langle *, s_i, sn, w \rangle\rangle$ is returned to server s_i. In this case, both server s_i's weight and the total weight of servers are not changed because the servers' weights are in a situation that Property 1 will be violated by returning $w' \neq 0$.

  – A change $\langle w' \neq 0, s_i, \langle *, s_i, sn, w \rangle\rangle$ is returned to server s_i. In this case, both server s_i's weight and the total weight of servers are increased by $w' = +w$ (resp. $w' = -w$) if $op = \mathsf{increase}$ (resp. $op = \mathsf{decrease}$).

One of the ways to implement this abstraction is to modify reconfiguration protocols (e.g., [6, 37, 38, 61]) in such a way that join and leave operations of reconfiguration protocols are modified to increase and decrease operations, respectively. An example of using this abstraction to reassign servers' weights is depicted in Figure 3.1. The decrease operation started by server s_2 is not finished because Property 1 will not be satisfied by executing this operation.

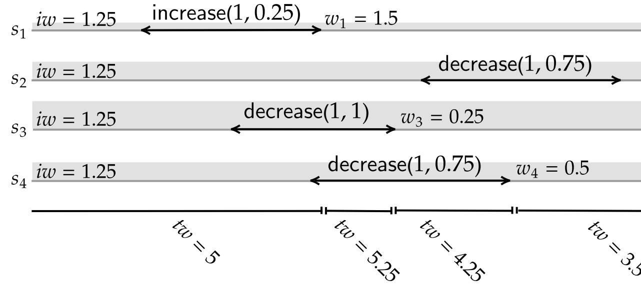


Figure 3.1: An example of the IncreaseDecrease weight reassignment abstraction

At first glance, it seems that this approach can be implemented using standard message-passing primitives [32]. However, the following theorem states

that this approach cannot be implemented in asynchronous, failure-prone distributed systems.

Theorem 1. The consensus number of IncreaseDecrease weight reassignment approach is at least two.

Proof. We prove the theorem by contradiction. For the sake of contradiction, we assume that the consensus number of the IncreaseDecrease approach is one (implicitly, we assume that the IncreaseDecrease approach can be implemented in failure-prone asynchronous distributed systems). Let $S = \{s_1, s_2, s_3\}$. We assume that every server executes Algorithm 3. Each server s_i, such that $i \in \{1, 2\}$, has access to a shared array of SWMR registers and writes its proposal in the ith register of the array. Each server s_i, such that $i \in \{1, 2\}$, calls an operation $op \in \{\mathsf{increase}, \mathsf{decrease}\}$ according to its identifier. Then, each server waits until the collect operation returns a value that is not equal to $\langle 1, 1, 1\rangle$. Finally, each server s_i, such that $i \in \{1, 2\}$, determines its decision based on the value returned by the called collect operation.

Algorithm 3 Solving consensus among two servers using the IncreaseDecrease approach

---

▷ s_1 and s_2 reach an agreement while s_3 is used as an auxiliary server
▷ the algorithm executed by each server s_i, where $i \in \{1, 2, 3\}$
▷ *proposals* is an array of SWMR registers to store s_1's and s_2's proposals such that $proposals[i]$ stores s_i's proposal

function decide()
1) $iw \leftarrow 1$
2) **if** $i = 1$
3) increase$(s_1, 1, 0.9)$
4) **else if** $i = 2$
5) decrease$(s_2, 1, 0.6)$
6) **wait until** $\langle 1, 1, 1\rangle \neq \mathsf{collect}() = \langle w_1, w_2, w_3\rangle$
7) **if** $w_1 = 1.9$
8) **return** $proposals[1]$
9) **else if** $w_2 = 0.4$
10) **return** $proposals[2]$

Note that line 6 will eventually be executed because we assume the IncreaseDecrease approach can be implemented in failure-prone asynchronous distributed systems. In total, there are four cases that only two of them can happen:

1) No operation is executed. This case cannot happen because we assumed that there is an asynchronous algorithm.

2) All operations are executed concurrently. Accordingly, the servers' weights become $\langle 1.9, 0.4, 1\rangle$. This case cannot happen since Property 1 is not satisfied.

3) Operation increase$(s_1, 1, 0.9)$ is executed. Accordingly, the value returned by function collect is $\langle 1.9, 1, 1\rangle$. This case can happen; as a result, the decided value by all servers is $proposals[1]$ (Line 8).

4) Operation decrease$(s_2, 1, 0.6)$ is executed. Accordingly, the value returned by function collect is $\langle 1, 0.4, 1\rangle$. This case can happen; as a result, the decided value by all servers is $proposals[2]$ (Line 10).

By using the aforementioned algorithm, the servers can solve consensus. Hence, there is a contradiction because three servers can solve consensus using the above algorithm, and the theorem holds. □

According to Theorem 1, two servers can solve consensus using the protocol of the IncreaseDecrease approach. Since consensus cannot be implemented in asynchronous, failure-prone distributed systems, the protocol of the IncreaseDecrease approach cannot be implemented in the asynchronous, failure-prone distributed systems. Note that if IncreaseDecrease weight reassignment approach, instead of presenting three operations, presents two operations– (a) either increase or decrease and (b) collect– the impossibility result still holds using the same proof as the proof of Theorem 1.

3.3 PerfectPairwise Approach

PerfectPairwise is a weight reassignment approach in which weights are reassigned in a pairwise manner where a server called *donor* donates some part of its weight to another server called *donee* that has better performance. In order to name this approach, the term 'perfect' is used to emphasize that if the donor's weight decreases by weight w in a donation, the donee's weight increases by the same amount of weight w. In this approach, the total weight of servers remains time-invariant in contrast to the IncreaseDecrease approach.

This approach provides the following operation:

– donate(sn, w, s_j) that can be called by each server that wants to be a donor and donates some part of its weight to another server s_j,

where sn is a local and monotonically increasing sequence number of the donor, and w is the value of donated weight. Such a sequence number is increased before calling the donate operation. Operation donate(sn, w, s_j) called by server s_i returns two changes– a change $\langle -w', s_i, \langle \text{DNT}, s_i, sn, w, s_j\rangle\rangle$ to s_i and another change $\langle w'', s_j, \langle \text{DNT}, s_i, sn, w, s_j\rangle\rangle$ to s_j, where w' and w'' might be either equal to 0 or w (Figure 3.2). The approach has the following properties:

– (PerfectPairwise Validity) Server s_i has called operation donate(sn, w, s_j) before it (resp. server s_j) receives change $\langle *, s_i, \langle \text{DNT}, s_i, sn, w, s_j\rangle\rangle$ (resp. change $\langle *, s_j, \langle \text{DNT}, s_i, sn, w, s_j\rangle\rangle$).

– (PerfectPairwise Termination) If server s_i calls operation donate(sn, w, s_j), it (resp. server s_j) receives a change $\langle *, s_i, \langle \text{DNT}, s_i, sn, w, s_j\rangle\rangle$ (resp. a change $\langle *, s_j, \langle \text{DNT}, s_i, sn, w, s_j\rangle\rangle$) eventually if it (resp. s_j) is a correct server.

- (PerfectPairwise Donate-Accuracy) If server s_i calls operation donate(sn, w, s_j), one of the following cases can happen:
  - If a change $\langle w' = 0, s_i, \langle \text{DNT}, s_i, sn, w, s_j \rangle \rangle$ is returned to server s_i, then a change $\langle w'' = 0, s_j, \langle \text{DNT}, s_i, sn, w, s_j \rangle \rangle$ is returned to s_j as well. In this case, the weights of servers s_i and s_j do not change because the s_j's weight is in a situation that Property 1 will be violated by returning $w' \neq 0$ and $w'' \neq 0$.
  - If a change $\langle w' = -w, s_i, \langle \text{DNT}, s_i, sn, w, s_j \rangle \rangle$ is returned to server s_i, then a change $\langle w'' = w, s_j, \langle \text{DNT}, s_i, sn, w, s_j \rangle \rangle$ is eventually returned to s_j. In this case, the weight of server s_i (resp. s_j) decreases (resp. increases) by w after receiving the change.

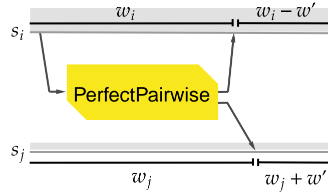


Figure 3.2: The illustration of the PerfectPairwise approach. Operation donate(sn, w, s_j) is called by server s_i, and two changes are returned to servers s_i and s_j. Consequently, the weight of server s_i (resp. s_j) decreases by w', where $w' = 0$ or w (resp. increases by w'', where $w'' = 0$ or w).

The following theorem shows that the PerfectPairwise approach cannot be implemented in a wait-free, failure-prone distributed systems, like IncreaseDecrease approach.

Theorem 2. The consensus number of PerfectPairwise approach is at least two.

Proof. We prove the theorem by contradiction. For the sake of contradiction, we assume that the consensus number of the PerfectPairwise approach is one (implicitly, we assume that an asynchronous algorithm can be presented for the PerfectPairwise approach). Let $S = \{s_1, s_2, s_3\}$. Using Algorithm 4, servers s_1 and s_2 reach an agreement, and server s_3 is used as an auxiliary server.

Each server s_i, such that $i \in \{1, 2\}$, has access to a shared array of SWMR registers and writes its proposal in the ith register of the array. Then, each server calls function decide. The initial weight of each server is equal to one. Each server s_i, such that $i \in \{2, 3\}$, calls the donate operation to donate 0.25 of its weight to s_1 (Lines 2-5). Then, all servers wait until receiving a change from the algorithm (Line 6). Finally, each server s_i, such that $i \in \{1, 2\}$, determines its decision based on the received change (Lines 7-16).

Since the donee is server s_1 in both donations, server s_1 receives two changes from the algorithm of PerfectPairwise approach that can be one of the following disjoint cases: (a) $\langle 0, s_1, \langle \text{DNT}, s_2, 1, 0.25, s_1 \rangle \rangle$ and $\langle 0.25, s_1, \langle \text{DNT}, s_3, 1, 0.25, s_1 \rangle,$

Algorithm 4 Solving consensus among two servers using the PerfectPairwise approach

▷ s_1 and s_2 reach an agreement while s_3 is used as an auxiliary server
▷ the algorithm executed by each server s_i, where $i \in \{1, 2, 3\}$
▷ *proposals* is an array of SWMR registers to store s_1's and s_2's proposals such that *proposals*$[i]$ stores s_i's proposal

function decide()
1) $iw \leftarrow 1$
2) **if** $i = 2$
3)   donate$(s_2, 1, 0.25, s_1)$
4) **else if** $i = 3$
5)   donate$(s_3, 1, 0.25, s_1)$
6) **wait until** receiving $\langle w, s_i, \langle \text{DNT}, s_k, 1, 0.25, s_1 \rangle\rangle$
7) **if** $i = 1$
8)   **if** $(k = 2$ **and** $w = 0)$ **or** $(k = 3$ **and** $w \neq 0)$
9)   **return** *proposals*$[1]$
10)   **else**
11)   **return** *proposals*$[2]$
12) **else if** $i = 2$
13)   **if** $w = 0$
14)   **return** *proposals*$[1]$
15)   **else**
16)   **return** *proposals*$[2]$

or (b) $\langle 0.25, s_1, \langle \text{DNT}, s_2, 1, 0.25, s_1 \rangle\rangle$ and $\langle\langle 0, s_1, \text{DNT}, s_3, 1, 0.25, s_1 \rangle$. The order of the received changes is not important.

On the other hand, server s_2 receives only one change that is either $\langle 0, s_2, \langle \text{DNT}, s_2, 1, 0.25, s_1 \rangle$ or $\langle -0.25, s_2, \langle \text{DNT}, s_2, 1, 0.25, s_1 \rangle$. If server s_1 receives one of the changes of case (a), then, it returns *proposals*$[1]$ (Line 9); server s_2 receives change $\langle 0, s_2, \langle \text{DNT}, s_2, 1, 0.25, s_1 \rangle$, then, it returns *proposals*$[1]$ as well (Line 14). Besides, if server s_1 receives one of the changes of case (b), then, it returns *proposals*$[2]$ (Line 11); server s_2 receives $\langle -0.25, s_2, \langle \text{DNT}, s_2, 1, 0.25, s_1 \rangle$, then, it returns *proposals*$[2]$ as well (Line 16). Accordingly, servers s_1 and s_2 can decide the same proposal, i.e., they can solve consensus. Hence, we found a contradiction, and the theorem holds. □

According to Theorem 2, two servers can solve consensus using the implementation of the PerfectPairwise approach. Since consensus cannot be implemented in asynchronous, failure-prone distributed systems, the presented implementation is not asynchronous.

As we have proved above, implementing IncreaseDecrease and PerfectPairwise approaches is impossible in a asynchronous, failure-prone distributed systems. Remember that the PerfectPairwise and IncreaseDecrease approaches are based on

TITW and TVTW, respectively. Since implementing TITW-based approaches is simpler than TVTW-based approaches, we present a weaker version of the PerfectPairwise approach, that can be implemented in asynchronous, failure-prone distributed systems. The weaker version is called the WeakPairwise approach.

3.4 WeakPairwise Approach

In the PerfectPairwise approach, a global computation should be performed for each donation (i.e., the total of the f greatest weights should be computed) to determine the changes returned to the donee and donor of the donation. To present a weaker version of the the PerfectPairwise approach that can be implemented in asynchronous, failure-prone distributed systems, we take the following steps:

- We modify the donate operation so that such a global computation is not required anymore. We define an upper bound for servers' weights denoted by $maxw$ such that $f \times maxw < tw/2$. By doing so, each server s_i returns $\min(maxw, w_i)$ as its weight. Hence, Property 1 is always satisfied.

  To indicate the value of $maxw$, we use the WHEAT scheme, i.e., we set $maxw = 1 + \Delta/f$ and $tw = 2 \times maxw \times f + 1$.

  In addition to defining an upper bound, it is required to define a lower bound for servers' weights denoted by $minw$. To see why $minw$ is required, consider the following example. Assume that $S = \{s_1, s_2, s_3, s_4\}$, $\Delta = 1$, and $f = 1$. Accordingly, $maxw = 2$, $tw = 5$, and $iw = 1.25$. Every server $s \in \{s_1, s_2, s_3\}$ donates weight 1 to server s_4. Consequently, the weights of servers s_1, s_2, s_3, and s_4 become 0.25, 0.25, 0.25, and 2, respectively. Then, server s_4 crashes. At this time, the system becomes unavailable. To avoid such a situation, the system should remain available until there exist $n - f$ servers, i.e., $tw/2 < (n - f) \times minw$. We can use the WHEAT scheme to define $minw$ as well, i.e., we set $minw = 1$.

- We assume that the initial weight, iw, of each server is equal to tw/n. Besides, we assume that the weight of each server can be any value in range $[minw, maxw]$.

- Each server is allowed to donate $iw - minw$ to other servers in total.

- We present another operation called retake by which a donee can retake its donated weight. To see why this operation is required, consider the following example. Assume that $S = \{s_1, s_2, s_3, s_4\}$, $\Delta = 1$, and $f = 1$. Accordingly, $maxw = 2$, $tw = 5$, $iw = 1.25$, and $minw = 1$. Every server $s \in \{s_1, s_2, s_3\}$ donates weight 0.25 to server s_4. Consequently, the weights of servers s_1, s_2, s_3, and s_4 become 1, 1, 1, and 2, respectively. Then, server s_4 crashes. At this time, servers cannot reassign theirs weights anymore. Theretofore, they need a mechanism to retake their donated weights. To do so, operation retake that is defined as follows is required.

- $\mathsf{retake}(sn, w, s_j)$ that can be called by each server that has donated some part of its weight to another server s_j and wants to retake its donated weight.

Operation $\mathsf{retake}(sn, w, s_j)$ called by server s_i returns two changes– a change $\langle w', s_i, \langle \text{RTK}, s_i, sn, w, s_j \rangle\rangle$ to s_i and another change $\langle -w'', s_j, \langle \text{RTK}, s_i, sn, w, s_j \rangle\rangle$ to s_j.

The weaker version of the PerfectPairwise approach is called the WeakPairwise approach and has the following properties:

- (PerfectPairwise Validity) The validity property has three parts:
  - Server s_i has called operation $\mathsf{donate}(sn, w, s_j)$ before s_i and s_j receive changes $\langle *, s_i, \langle \text{DNT}, s_i, sn, w, s_j \rangle\rangle$ and $\langle *, s_j, \langle \text{DNT}, s_i, sn, w, s_j \rangle\rangle$, respectively.
  - Server s_i has called operation $\mathsf{retake}(sn, w, s_j)$ before s_i and s_j receive changes $\langle *, s_i, \langle \text{RTK}, s_i, sn, w, s_j \rangle\rangle$ and $\langle *, s_j, \langle \text{RTK}, s_i, sn, w, s_j \rangle\rangle$, respectively.
  - Before calling operation $\mathsf{retake}(sn, w, s_j)$ by server s_i, server s_i has called operation $\mathsf{donate}(sn, w, s_j)$.
- (PerfectPairwise Termination) If server s_i calls operation $op(sn, w, s_j)$ such that $op \in \{\mathsf{donate}, \mathsf{retake}\}$, it (resp. server s_j) receives a change $\langle *, s_i, \langle *, s_i, sn, w, s_j \rangle\rangle$ (resp. a change $\langle *, s_j, \langle *, s_i, sn, w, s_j \rangle\rangle$) eventually if it (resp. s_j) is a correct server.
- (WeakPairwise Donate-Accuracy) If server s_i calls operation $\mathsf{donate}(sn, w, s_j)$, one of the following cases can happen:
  - If a change $\langle w' = 0, s_i, \langle \text{DNT}, s_i, sn, w, s_j \rangle\rangle$ is returned to server s_i, then a change $\langle w'' = 0, s_j, \langle \text{DNT}, s_i, sn, w, s_j \rangle\rangle$ is returned to s_j as well. In this case, the weights of servers s_i and s_j do not change because the s_j's weight is in a situation that Property 1 will be violated by returning $w' \neq 0$ and $w'' \neq 0$.
  - If a change $\langle w' = -w, s_i, \langle \text{DNT}, s_i, sn, w, s_j \rangle\rangle$ is returned to server s_i, then a change $\langle w'', s_j, \langle \text{DNT}, s_i, sn, w, s_j \rangle\rangle$ is eventually returned to s_j, where w'' is equal to $\mathsf{min}(maxw - w_j, w)$. In this case, the weight of server s_i decreases by w, and the weight of server s_j increases by $\mathsf{min}(maxw - w_j, w)$, after receiving the changes. In other words, operation $\mathsf{donate}(sn, w, s_j)$ called by server s_i works as follows:

```
1) if minw ≤ w_i − w
2)     w_i ← w_i − w
3)     w_j ← min(maxw, w_j + w)
4) other servers' weights are not changed
```

– (PerfectPairwise Retake-Accuracy) Given operation retake(sn, w, s_j) called by server s_i, a change $\langle w', s_i, \langle \text{RTK}, s_i, sn, w, s_j \rangle\rangle$ is returned to server s_i, and another change $\langle w'' = -w, s_j, \langle \text{RTK}, s_i, sn, w, s_j \rangle\rangle$ is eventually returned to s_j, where $w' = \mathsf{min}(maxw - w_j, w)$ In this case, the weight of server s_i (resp. s_j) increases (resp. decreases) by $\mathsf{min}(maxw - w_j, w)$ (resp. w) after receiving the change.

In a donation made from server s_i to server s_j, some part of the donated weight might be lost. To grasp the idea of losing weight, consider the following example. Let $S = \{s_1, \ldots, s_6\}$ and $f = 2$. Consequently, $tw = 7$, $iw = 7/6$, and $maxw = 1.5$. Assume that each server $s_i \in \{s_2, \ldots, s_6\}$ donates a weight equal to $1/6$ to server s_1. Besides, assume that such donations are concurrent. After applying the donations, the weights of servers are reassigned to the values determined in Figure 3.3, and the summation of weights is equal to 6.5, which means that 0.5 part of tw is lost. Note that losing weights enables the WeakPairwise approach to be implemented in asynchronous, failure-prone distributed systems.

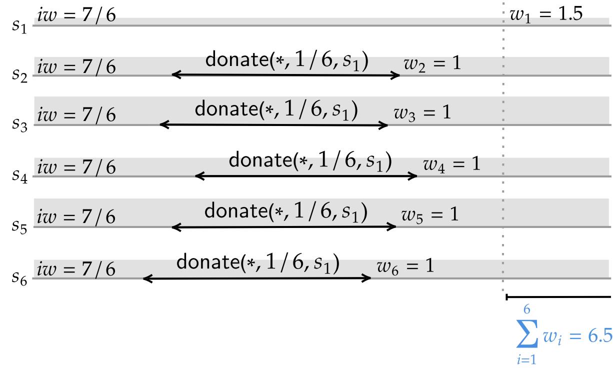


Figure 3.3: An example of losing weight in the WeakPairwise approach

3.5 Discussion

This section consists of two distinct parts. Due to the similarity of the weight reassignment and reconfiguration problems, the first part is dedicated to presenting the impossibility result of the IncreaseDecrease and PerfectPairwise approaches for the reconfiguration problem. The second part discusses the relationship between the weight reassignment and asset transfer problems.

3.5.1 Weight Reassignment vs. Reconfiguration

The reconfiguration problem is the problem of changing the set of servers over time. Reconfiguration protocols (e.g., [5, 6, 38]) provides two operations– join

and leave. Let U be the universe of servers such that $S \subseteq U$. Each server $s \in U$ can join to the system, i.e., can be added to set S, by calling join(s). Besides, each server $s \in U$ can leave the system, i.e., can be removed from set S, by calling leave(s). In the following, we present the relationship between the weight reassignment and reconfiguration problems. To do so, we show that the reconfiguration problem is a special case of the IncreaseDecrease weight reassignment approach.

Remember that the IncreaseDecrease approach provides two operations– increase and decrease– to reassign the weights of servers. We present the following method to simulate operation join (resp. leave) with operation increase (resp. decrease):

- If a server $s \in U$ is a member of the initial configuration, its weight is equal to one; otherwise, its weight is equal to zero.

- Simulating operation join with operation increase: when a server $s \in U$ wants to join the system, it calls operation increase$(1, 1)$.

- Simulating operation leave with operation decrease: when a server $s \in U$ that is a member of the system wants to leave the system, it calls operation decrease$(1, 1)$.

The reason why the IncreaseDecrease approach has no asynchronous implementation is that servers might concurrently execute increase and decrease operations such that Property 1 is not satisfied. Based on this reason and the relationship between the IncreaseDecrease approach and reconfiguration problem, we define a variant of reconfiguration problem called *conditional asynchronous reconfiguration*. The *conditional asynchronous reconfiguration* problem is similar to the standard asynchronous reconfiguration problem with only one difference: a condition related to the size of configurations should be satisfied. For example, the size of each configuration should be greater than k, where $2 \leq k \leq n$. We prove that solving such a problem is impossible in asynchronous, failure-prone distributed systems. To do so, we show that the consensus number of the *conditional asynchronous reconfiguration* problem is at least three.

Theorem 3. The consensus number of the *conditional asynchronous reconfiguration* problem is at least three.

Proof. We prove the theorem by contradiction. For the sake of contradiction, we assume that the consensus number of the *conditional asynchronous reconfiguration* problem is one (implicitly, we assume that there is an algorithm that solves the *conditional asynchronous reconfiguration* problem in asynchronous, failure-prone distributed systems).

Let $S = \{s_1, s_2, s_3\}$ be the current configuration. Also, let $|C| \geq 2$ be the condition that should be satisfied in every configuration C. Such a condition states that the number of servers in each configuration C should be greater than equal to two. Assume that three concurrent *leave* requests– leave(s_1), leave(s_2),

Algorithm 5 Solving consensus among two servers using the implementation of the *conditional asynchronous reconfiguration* problem

▷ the algorithm executed by each server s_i, where $i \in \{1, 2, 3\}$
▷ *proposals* is an array of SWMR registers to store servers' proposals such that *proposals*[*i*] stores s_i's proposal

function decide()
1) $cur_conf \leftarrow \{\langle +s_1 \rangle, \langle +s_2 \rangle, \langle +s_3 \rangle\}$
2) $proposal_conf \leftarrow cur_conf \cup \{\langle -s_i \rangle\}$
3) execute the algorithm of the *conditional asynchronous reconfiguration* problem with input $proposal_conf$
4) **wait until** a new configuration is returned
▷ assume that the new configuration is new_conf
5) **find** k such that $s_k \notin new_conf$
6) **return** $proposals[k]$

and leave(s_3)– are issued by different clients. By using Algorithm 5, servers can solve consensus.

Note that lines 5-6 will eventually be executed because we assumed that there is an algorithm for solving the *conditional asynchronous reconfiguration* problem in asynchronous, failure-prone distributed systems. Besides, note that the size of new_conf is two due to the following reasons: (a) the size of new_conf cannot be less than two because the defined condition will not be satisfied, and (b) the size of new_conf cannot be three because each server waits until having a new configuration (line 4). There is a contradiction because three servers can solve consensus using the above algorithm, and the theorem holds. □

Corollary 1. Solving the *conditional asynchronous reconfiguration* problem in asynchronous, failure-prone distributed systems is impossible.

Proof. This corollary holds according to Theorem 3. □

3.5.2 Weight Reassignment vs. Asset Transfer

Each server s_i has an account in the asset transfer problem and can transfer some part of its balance to another server s_j if the transferred balance does not turn its balance negative, formally:

1) **if** $0 \leq w_i - w$
2) $w_i \leftarrow w_i - w$
3) $w_j \leftarrow w_j + w$
4) other servers' weights are not changed

where w_k and w denote the balance of server s_k and the transferred balance, respectively. The following two facts and a conclusion that comes later make it

more clear what makes implementing the PerfectPairwise approach to be impossible in asynchronous, failure-prone distributed systems. (a) the asset transfer problem is similar to the weight reassignment problem with only one difference: there is no upper bound for the balance (weight) of each server, and (b) it is proved that the asset transfer problem can be implemented in asynchronous, failure-prone distributed systems [28]. The conclusion that can be made based on the presented facts: satisfying the upper bounds of servers' weights makes it impossible to implement the PerfectPairwise approach in asynchronous, failure-prone distributed systems.

Chapter 4

Epoch-Based Weight Reassignment

Abstract. This chapter presents a crash fault-tolerant (CFT) epoch-based weight reassignment protocol for implementing the WeakPairwise weight reassignment approach in asynchronous distributed systems. In such a protocol, the chief duty of epochs is to determine the lifetimes of changes returned by donate and retake operations. Then, an atomic storage based on DWMQS is designed such that each server has a weight, the quorums are constituted using servers' weights, and servers' weights can be reassigned using the proposed weight reassignment protocol over time.

4.1 Warmup

Remember that WeakPairwise is a weight reassignment approach, that provides two operations– donate and retake. Each server can donate some part of its weight to another server and retake the donated weight by calling donate and retake operations, respectively. Operations donate and retake create changes, and every created change is added to set *changes*. The weight of each server can be computed using set *changes*. This chapter proposes a protocol to implement donate and retake operations.

The proposed protocol is epoch-based. The system's lifetime is divided into a sequence of epochs $\langle e_0, e_1, \dots \rangle$. At the beginning, epoch e_0 is installed in the system. After finishing each epoch e_k $(0 \leq k)$, epoch e_{k+1} is installed. The chief duty of epochs is to determine the lifetimes of changes created by donate and retake operations as follows.

The default weight of each server during each epoch is equal to iw. Before installing epoch e_k, each server s_i can donate some part of its default weight associated with epoch e_k to another server s_j. To do so, server s_i sends the amount of weight that wants to donate for epoch e_k within a DNT message to server s_j. Server s_j receives the DNT message if it is a correct server. The donate operation completes when epoch e_{k-1} is changed to epoch e_k such that server s_i (resp. server s_j) creates a change $\langle -w, s_i, * \rangle$ (resp. a change $\langle w, s_j, * \rangle$)

and adds the created change to set *changes*. The lifetimes of both changes is epoch e_k. Indeed, the retake operation is executed at the time of changing epoch e_k to epoch e_{k+1}, and s_i (resp. s_j) creates a change $\langle w, s_i, * \rangle$ (resp. a change $\langle -w, s_j, * \rangle$), and adds the created change to set *changes*. The whole process is illustrated in Figure 4.1.

There are two algorithms by which the donate and retake operations are implemented. The first algorithm called PairwiseWR is executed by each server and is responsible for finding servers as donees, sending DNT messages to donees, and processing the received DNT messages. The other algorithm called EpochChanger is executed by servers as well and is responsible for changing the epochs, completing the executions of donate operations, and executing the retake operations.

In the following, we explain the epoch-based protocol in further details. First, we present a few preliminary definitions and concepts in Section 4.2. Then, we explain the PairwiseWR algorithm. Then, we present the EpochChanger algorithm. We present the read-write protocols and an example containing a few donations, retakes, and clients' read-write requests.

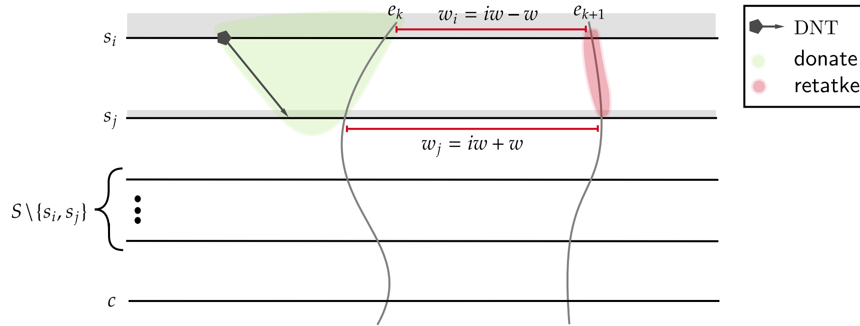


Figure 4.1: Executions of the donate and retake operations

4.2 Preliminaries

In this section, we present the preliminary definitions and concepts related to the epoch-based weight reassignment protocol.

Epochs. In the epoch-based weight reassignment protocol, the object of epochs is to determine the following items:

- the lifetimes of changes returned by donate and retake operations, i.e., when the changes returned by such operations should be applied to the weights of donors and donees, and
- the execution time of the retake operations.

During the system's lifetime, a sequence of epochs $\sigma = \langle e_0, e_1, \dots \rangle$ is installed in the system. The following assumption determines the number of epochs

installed in the system. A similar assumption is used by all protocols that implement the reconfigurable atomic storage such as RAMBO [25], DynaStore [5], SpSn [20], FreeStore [6], and SmartMerge [38]. Its reason is that it is impossible to reconfigure a storage system infinitely many times while guaranteeing the liveness of the storage system [59].

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

The system starts in epoch e_0 called the *initial epoch*. The successor (resp. predecessor) of any epoch e_k for $0 \leq k$ (resp. $1 \leq k$) is $e_{k+1} = e_k.succ$ (resp. $e_{k-1} = e_k.pred$). When a non-initial epoch e_{k+1} $(0 \leq k)$ is installed, we say that $e_{k+1}.pred$ was uninstalled from the system. At any time $t \in \mathcal{T}$, we define *lepoch* to be the last epoch installed in the system. Since *lepoch* is the last epoch installed in the system, it does not have any successor, that is shown by $lepoch.succ = \perp$. The weights of servers are not reassigned during any epoch and might be reassigned at the time of installing epochs.

Each process $p \in \{C \cup S\}$ has a local variable called *cepoch* to store its current epoch. The current epoch of each process p, $p.cepoch$, is a member of set $\{e_0, e_1, \ldots, lepoch\}$. Note that the current epochs of any two processes might be different, and also, they might be different from the last installed epoch in the system.

Installing an epoch. To install an epoch in the system, at least one server should request it. Each server s_i can only request to install epoch $s_i.cepoch.succ$ in the system by executing the EpochChanger algorithm. If $lepoch = s_i.cepoch$, the EpochChanger algorithm installs epoch $s_i.cepoch.succ$ in the system such that the current epochs of a weighted majority of servers are changed to epoch $s_i.cepoch.succ$.

Comparing two epochs. We say that epoch e' is *more up-to-date* than epoch e if the following recursive function returns *yes* by passing (e, e') as input. We use the notation $e < e'$ to state that epoch e' is *more up-to-date* than epoch e.

```
function more_up_to_date(e, e')
  e ← e.succ
  if e = e' then return yes
  else if e = ⊥ then return no
  else return more_up_to_date(e, e')
```

Epochs vs. read-write operations. At any time $t \in \mathcal{T}$, read-write operations can only be executed in epoch *lepoch*. At the time of uninstalling any epoch e, read-write operations are disabled on every server s such that $s.cepoch = e$. The operations are enabled after installing epoch $e.succ$.

Change. Each change made on a server s_i's weight is associated with an epoch e and has the following structure: $\langle \pm w, s_i, \langle type, e, info \rangle \rangle$, where w is the value of the change, $type \in \{\text{INIT}, \text{DNT}, \text{RTK}\}$, and $info$ might contain additional

information about the change. Types INIT, DNT, and RTK are used when a change is added as the initial weight of a server, returned from the donate operation, and returned from the retake operation, respectively. Each returned change is added to set *changes*.

Calculating weights. The weight of each server s_i for each epoch e_k can be calculated by calling function weight(s_i, e_k) as follows: the total weight of all changes created for epoch e_k is computed; if the computed weight is less than equal to $maxw$, then it is returned; otherwise, $maxw$ is returned.

function weight(s_i, e_k)
  $w \leftarrow \mathsf{sum}\big(\{w \mid \langle w, s_i, \langle *, e_k, * \rangle\rangle \in s_i.changes\}\big)$
  return min$(w, maxw)$

Quorum. Processes can use function is_quorum to determine whether a subset of servers S' constitutes a quorum in epoch e_k. This function computes the total weight of all servers S' using function weight. If the computed total weight is greater than $tw/2$, subset S' constitutes a quorum. This function works as follows in further details.

function is_quorum(S', e_k)
1) **if** $tw/2 < \mathsf{sum}\big(\{\mathsf{weight}(s_i, e_k) \mid s_i \in S'\}\big)$
2)   **return** *yes*
3) **else**
4)   **return** *no*

4.3 PairwiseWR Algorithm

This section describes the PairwiseWR algorithm of the epoch-based weight reassignment protocol. The PairwiseWR algorithm is executed by each server and is responsible for finding servers as donees, sending DNT messages to donees, and processing the received DNT messages. The default weight of servers in each epoch is equal to iw, that can be calculated using the Extended WHEAT scheme.

Algorithm 6 is the pseudocode of the PairwiseWR algorithm. Each server has a set denoted by *changes* to store every received change initialized with a change $\langle iw, s_i, \langle \text{INIT}, e_0 \rangle\rangle$ (Line 1 of Algorithm 6). Also, each server has a variable called sn initialized with 0 and used as a sequence number to distinguish its donations (Line 2 of Algorithm 6). Each server has access to three functions: • weight that returns the weight of a server for a specific epoch, • cepoch that returns the current epoch of a server, and • dirty_epochs that returns a set of epochs that a server has participated in.

For each epoch, any server might reassign its default weight. As a result, each server might have different weights in distinct epochs. Each server s_i to reassign its default weight associated with an epoch e, such that $s_i.cepoch < e$,

should donate some part of its weight or receive at least one donation. In this way, the servers that have lower latency scores might become high-weighted servers leading to improving the performance.

Each server should satisfy the lower bound defined for weights in Assumption 2 to be allowed to donate some weight. For simplicity, each server s_i can only donate some weight for its succeeding epoch (epoch $s_i.cepoch.succ$). Server s_i can donate weight w associated with epoch e to another server s_j if the following conditions are satisfied:

C1) Epoch e should be equal to the succeeding epoch of server s_i.

C2) Server s_i has not participated in any operation related to epoch e, i.e., $e \notin dirty_epochs$.

C3) Server s_i's latency score should be significantly greater than s_j's latency score, i.e., $\mathsf{lscore}(s_j) \ll \mathsf{lscore}(s_i)$. Note that the reason for not using 'greater than' and using 'significantly greater than' is to avoid unnecessary weight reassignments.

C4) Server s_i should have some weight to donate to server s_j. Each server s_i is allowed to donate at most $iw - minw$ part of its weight. Indeed, server s_i is allowed to donate weight w to another server s_j if the following relation holds: $iw - minw \leq \mathsf{weight}(s_i, e) - w$.

If the above conditions are met, server s_i is allowed to donate weight w to server s_j. To do so, server s_i takes the following steps: (a) increments its sequence number, (b) creates a change for the donation, (c) adds the created change to set *chagnes*, and (d) sends a message $\langle \text{DNT}, cepoch.succ, s_i, sn, w, s_j \rangle$ to sever s_j (Lines 6-8 of Algorithm 6). After receiving a donation, each server s_i creates a change for the donation if the following condition is met:

C5) Server s_i has not participated in any operation related to epoch e.

If the above conditions are met, server s_i creates a change for the received donation and adds the created change to set *changes* (Line 12 of Algorithm 6). Otherwise, it does not create a change for the received donation.

4.4 EpochChanger Algorithm

Each server can request to change the installed epoch in the system and change its current epoch by using an algorithm called the EpochChanger algorithm. Algorithm 7 is the pseudo-code of the EpochChanger algorithm. In the following, we describe how this algorithm works.

How servers can request to change an epoch. Each server has a variable called *cepoch* to store its current epoch (Line 1 of Algorithm 7). Function cepoch returns the value of this variable. Each server s_i for each epoch e has a timeout. Note that in practice, such a timeout should be big enough so that the

Algorithm 6 The PairwiseWR algorithm - server s_i

variables
1) $changes \leftarrow \{\langle iw, s_i, \langle \text{INIT}, e_0\rangle\rangle\}$
2) $sn \leftarrow 0$

while $forever$
3) $cepoch \leftarrow$ cepoch() of Algorithm 7
4) $dirty_epochs \leftarrow$ dirty_epochs() of Algorithm 7
5) **if** $cepoch.succ \notin dirty_epochs$ **and** $\exists\ s_j \neq s_i$: $\mathsf{lscore}(s_j) \ll \mathsf{lscore}(s_i)$
   and $minw \leq \mathsf{weight}(s_i, cepoch.succ) - w$
6) $sn \leftarrow sn + 1$
7) $changes \leftarrow changes \cup \{\langle -w, s_i, \langle \text{DNT}, cepoch.succ, s_i, sn, w, s_j\rangle\rangle\}$
8) **send** $\langle \text{DNT}, cepoch.succ, s_i, sn, w, s_j\rangle$ to s_j

upon receipt of $\langle \text{DNT}, epoch, s_j, s, w, s_i\rangle$ from s_j
9) $cepoch \leftarrow$ cepoch() of Algorithm 7
10) $dirty_epochs \leftarrow$ dirty_epochs() of Algorithm 7
11) **if** $epoch \notin dirty_epochs$
12) $changes \leftarrow changes \cup \{\langle w, s_i, \langle \text{DNT}, epoch, s_j, s, w, s_i\rangle\rangle\}$

epochs are changed rarely to satisfy Assumption 4. When the timeout of epoch $s_i.cepoch$ finishes, server s_i broadcasts a message $\langle \text{CHANGE}, s_i.cepoch.succ\rangle$ to all servers to change epoch $s_i.cepoch$ to epoch $s_i.cepoch.succ$. Then, it stores $s_i.cepoch.succ$ in a set denoted by $s_i.change_requests$ (Lines 9-10 of Algorithm 7). Set $change_requests$ is used to change epochs.

How a server can change its current epoch. Each server s_i, by receiving any message $\langle \text{CHANGE}, epoch\rangle$, stores $epoch$ in a set denoted by $s_i.change_requests$ (line 25 of Algorithm 7). As soon as $s_i.cepoch.succ \in s_i.change_requests$, server s_i starts to change its epoch (line 13). To do so, server s_i must take the following steps: (a) Removing the timeout dedicated to epoch $epoch$ if it has not been removed or finished yet (Lines 12-13 of Algorithm 7). (b) Disabling read-write operations (Line 14 of Algorithm 7). (c) Informing Algorithm 6 that some operations related to epoch $s_i.cepoch.succ$ are processing to safety Conditions C2 (Line 15 of Algorithm 7). (d) Updating the states (registers) of servers (Lines 16-21 of Algorithm 7). Each server has a register; in this step, the registers of servers with epoch $epoch$ are synchronized. To do so, server s_i reads its register, $\langle tag, val\rangle$ (Line 16). Then, server s_i broadcasts message $\langle \text{UPDATE}, s_i, \langle tag, val\rangle, s_i.cepoch, s_i.cepoch.weight\rangle$ to other servers (Line 17). Server s_i waits until receiving messages $\langle \text{UPDATE}, *, *, s_i.cepoch, *\rangle$ from a weighted majority of servers (Line 19). Server s_i computes and stores the new state of its register (Lines 20-22). (e) Changing the epoch (Line 23). (f) Enabling read-write operations (Line 24).

Example 2. Figure 4.2 illustrates an example of executing the PairwiseWR and the EpochChanger algorithms. In this example, $S = \{s_1, s_2, s_3, s_4\}$, $f = 1$, and $\Delta = 1$. Using WHEAT scheme, $iw = 1.25$, $wmax = 2$, and $wmin = 1$. Server s_3 wants to donate weight 0.25 associated with epoch e_1 to server s_1. Hence, server

Algorithm 7 The EpochChanger algorithm - server s_i

variables

1) $cepoch \leftarrow e_0$
2) $change_requests \leftarrow \varnothing$
3) $updates \leftarrow \varnothing$
4) $dirty_epochs \leftarrow \{e_0\}$

functions

5) $\mathsf{maxtag}(e) \equiv \mathsf{max}\big(\{tag \mid (*, \langle tag, * \rangle, epoch, *) \in updates \text{ and } e = epoch\}\big)$
6) $\mathsf{maxval}(e, t) \equiv \mathsf{first}\big(\{val \mid (*, \langle tag, val \rangle, epoch, *) \in updates \text{ and } e = epoch \text{ and } t = tag\}\big)$
7) $\mathsf{cepoch}() \equiv cepoch$
8) $\mathsf{dirty_epochs}() \equiv dirty_epochs$

upon timeout for epoch *cepoch*

9) **RB_broadcast** $\langle \text{CHANGE}, cepoch.succ \rangle$
10) $change_requests \leftarrow change_requests \cup \{epoch\}$

while *forever*

11) **if** $cepoch.succ \in change_requests$
12) **if** $\exists$ timeout for epoch *cepoch*
13) remove the timeout
14) **disable** the execution of read-write operations of Algorithm 9
15) $dirty_epochs \leftarrow dirty_epochs \cup \{cepoch.succ\}$
16) $\langle tag, val \rangle \leftarrow \mathsf{register}()$ of Algorithm 9
17) **RB_broadcast** $\langle \text{UPDATE}, s_i, \langle tag, val \rangle, cepoch, cepoch.weight \rangle$
18) $updates \leftarrow updates \cup \{(s_i, \langle tag, val \rangle, cepoch, cepoch.weight)\}$
19) **wait** until $\mathsf{is_quorum}(updates)$
20) $maxtag \leftarrow \mathsf{maxtag}(cepoch)$
21) $val \leftarrow \mathsf{maxval}(cepoch, maxtag)$
22) $\mathsf{set_tagval}(maxtag, val)$ of Algorithm 9
23) $cepoch \leftarrow cepoch.succ$, set a timeout for *cepoch*
24) **enable** the execution of read-write operations

upon RB_deliver $\langle \text{CHANGE}, e \rangle$

25) $change_requests \leftarrow change_requests \cup \{e\}$

upon RB_deliver $\langle \text{UPDATE}, s, \langle tag, val \rangle, epoch, weight \rangle$

26) $updates \leftarrow updates \cup \{(s, \langle tag, val \rangle, epoch, weight)\}$

s_3 sends a DNT message to server s_1 during epoch e_0. Furthermore, server s_4 wants to donate weight 0.25 associated with epoch e_1 to server s_1. Server s_4 sends a DNT message to server s_1 during epoch e_0 as well. Then, server s_1 receives the DNT messages sent by servers s_3 and s_4 before installing epoch e_1. Therefore, server s_1 creates two changes for the received donations. The new weights of servers s_1, s_2, s_3, and s_4 in epoch e_1 are 1.75, 1.25, 1, and 1, respectively.

Server s_4 wants to donate weight 0.25 associated with epoch e_2 to server s_2. Consequently, server s_4 sends a message DNT to server s_2. Server s_2 receives

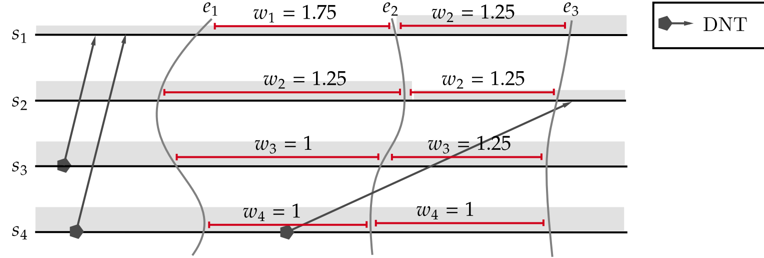


Figure 4.2: An example of executing executing the PairwiseWR and the EpochChanger algorithms. The weight of each server in epoch e_1 (resp. epoch e_2) is shown at the top of epoch e_1 (resp. epoch e_2).

this message during epoch e_3. Due to Condition C5, server s_2 has participated in epoch e_2, so server s_2 does not create a change for the donation while server s_4 creates a change. As a result, the weights of servers s_2 and s_4 in epoch e_2 are equal to 1.25 and 1, respectively.

Properties of Epoch Changer

We present properties of Algorithm 7 in this subsection. These properties are used to prove that: (1) for each non-initial epoch e installed in the system, there exists at least one server that requests to install epoch e, and (2) the weight reassignment algorithm satisfies the liveness property. Moreover, these properties are used in Section 4.5 to prove the correctness of the atomic storage. Note that all the specified lines in this subsection are related to Algorithm 7.

Lemma 1. *Let epoch e be the current epoch of a server s, i.e., $s.cepoch = e$. Server s installs epoch $e.succ$ if at least a weighted majority of servers including s had uninstalled epoch e.*

Proof. To install epoch $e.succ$, server s requires executing Line 23. To do so, first, it should receive message $\langle \text{CHANGE}, e.succ \rangle$. Then, it can execute Lines 13-17. If server s passes Line 19, it means that s received messages tagged with UPDATE from a weighted majority of servers ρ. Each server $s' \in \rho$ has uninstalled epoch e before sending a message tagged with UPDATE. Therefore, server s can be sure that at least a weighted majority of servers has uninstalled epoch e by passing Line 19. Then, server s can continue executing the remaining lines to change its epoch to $e.succ$. □

Lemma 2. *There is only one installed epoch in the system. In other words, if an epoch e' is installed in the system, any previously installed epoch $e < e'$ was uninstalled and will not be installed anymore.*

Proof. We will argue by contradiction. Assume that there are at least two installed epochs e and e' in the system, where $e \neq e'$. Without loss of generality

we can assume $e < e'$. To install e' in the system, according to the definition of installing a epoch, at least one server s should install epoch e'. Due to Lemma 1, at least a weighted majority of servers with epoch $e'.pred$ including s had uninstalled epoch $e'.pred$ to allow s for installing epoch e'. Similarly, we can show that at least a weighted majority of servers with epoch $e'.pred.pred$ including s had uninstalled epoch $e'.pred.pred$. With the same argument, we can show that any epoch x less than e' had uninstalled by at least a weighted majority of servers with epoch x including s. Therefore, e was uninstalled by at least a weighted majority of servers with epoch e.

Assume that e was the ith installed epoch in the system and e' is the jth installed epoch in the system, where $i < j$. To install a new epoch after epoch e', the index of the new epoch should be greater than j. Therefore, epoch e will not be installed anymore. □

Lemma 3. *Let s and s' be two correct servers such that: (1) $s.cepoch = e_j$, (2) $s'.cepoch = e_i$, (3) $e_i < e_j$, (4) after uninstalling epoch e_i, the sequence of epochs installed by server s is $\langle e_{i+1}, \ldots, e_{j-1}, e_j \rangle$, and (5) the current epoch of server s' is less than equal to the current epochs of all correct servers. Server s' installs the same sequence of epochs $\langle e_{i+1}, \ldots, e_{j-1}, e_j \rangle$ eventually.*

Proof. When $s.cepoch = e_i$, server s requires executing Line 13-23 to install epoch e_{i+1}. By executing those lines, it is guaranteed that two messages are sent by server s: message $\langle \text{CHANGE}, e_{i+1} \rangle$ and message $\langle \text{UPDATE}, *, *, *, e_i, * \rangle$. In addition to server s', there might be some other servers with epoch e_i. Since server s is correct, all correct servers including server s' with epoch e_i receive message $\langle \text{CHANGE}, e_{i+1} \rangle$ eventually. To finish the proof, we need to show the following cases.

1) All correct servers including server s' with current epoch e_i uninstall epoch e_i and send message $\langle \text{UPDATE}, *, *, *, e_i, * \rangle$.

2) All correct servers including server s' with current epoch e_i can only change their epochs to e_{i+1}.

3) Server s' can pass Line 19, i.e., server s' receives messages $\langle \text{UPDATE}, *, *, *, e_i, * \rangle$ from a weighted majority of servers (then server s' can continue executing Lines 20-23 to change its epoch to e_{i+1}.)

The first two cases are straightforward. Regarding the third case, according to Property 1, even if f servers fail (each of them with weight $maxw$), there might be a weighted majority of correct servers that sent message $\langle \text{UPDATE}, *, *, *, e_i, * \rangle$. Therefore, the last case eventually occurs. □

Lemma 4. *Let $lepoch = e_k$ at time t such that e_k is the $k + 1^{th}$ epoch installed in the system, where $0 \leq k$. There is only one sequence of epochs $\sigma = \langle e_0, e_1, \ldots, e_{k-1}, e_k \rangle$ from e_0 to e_k such that $e_i = e_{i-1}.succ$ for any $1 \leq i \leq k$.*

Proof. Using Lemmas 2 and 3, the proof of this lemma is straightforward. □

Lemma 5. *Let epoch e be the last installed epoch in the system, i.e., $lepoch = e$. Assuming a correct server s requests to change epoch e to epoch $e.succ$, then at least a weighted majority of correct servers including s install $e.succ$ eventually.*

Proof. Using Lemma 3, all correct servers will eventually reach to epoch e. Then, according to Property 1, the total weight of correct servers with epoch e is a weighted majority. If a correct server (say, s) with epoch e sends a request to change its epoch to epoch $e.succ$, it will change its epoch due to the existence of a weighted majority of correct servers with epoch e. After that, all correct servers will eventually reach to epoch $e.succ$ by Lemma 3, and the total weight of correct servers with epoch e is a weighted majority according to Property 1. ☐

Lemma 6. *Let e be the last installed epoch in the system. Epoch e will eventually be changed.*

Proof. Due to Lemma 5, there exists (or will eventually exist) at least a weighted majority of correct servers with epoch e. The timeout of at least one of those servers (say, s) for epoch e will eventually finish; then s will send message $\langle \text{CHANGE}, e.succ \rangle$ to other servers. Since server s is correct, the epoch will eventually be changed due to Lemma 5. ☐

Theorem 4. The algorithm is live, i.e., servers can change their weights over time.

Proof. Using Lemmas 5 and 6, the proof of this theorem is straightforward. ☐

4.5 Read-Write Protocols

We extend the (static) MW-ABD protocol [45] in such a way that each server has a weight, the quorums are constituted using servers' weights, and servers' weights can be reassigned using the proposed epoch-based weight reassignment protocol over time. In the following, we present the reader-writer and server sides of read-write protocols.

Reader-writer side. Algorithm 8 describes the reader-writer side of the read-write protocols. Each process has a variable *cepoch* to store its current epoch initialized with epoch e_0 and has a variable *cnt* used as a sequence number to distinguish the issued requests initialized with zero (Line 1 of Algorithm 8). Both read and write operations proceed in two phases, each ending with a confirmation that at least one quorum was accessed (like the original MW-ABD protocol).

In the first phase of a read or write operation (phase1), a process contacts a quorum in order to determine the last epoch installed in the system and highest tag *maxtag* used prior to write the last written value. To do so, the process sends a message $\langle \text{READ}, cnt \rangle$ to all servers. Servers include their current epochs in their responses. The process stores all received responses that their epochs are equal to its current epoch and can determine whether a subset of servers

constitutes a quorum using function is_quorum. If the current epoch of the process is less up-to-date than the epoch of a response, the process updates its current epoch with that epoch and restarts the operation (Lines 12-14 of Algorithm 8).

Then, in the second phase (phase2), after incrementing the timestamp of *maxtag*, if the process is a writer, it overwrites its own identifier on the identifier of *maxtag* and propagates its current epoch, value, and the changed *maxtag* within a WRITE message to a quorum.

Server side. Algorithm 9 is the pseudo-code of the server side of read-write protocols. After receiving each read-write request r, each server s determines its current epoch ($s.cepoch$) by calling function cepoch() from Algorithm 7 and computes its weight using function weight (Lines 5-6 of Algorithm 9). For write requests, if the current epoch of each request is the same as $s.cepoch$, server s executes the request (Lines 11-12 of Algorithm 9). Similarly, for read requests, server s adds $s.cepoch$ and *weight* to its response that should be sent to the client that issued the request.

Correctness

In the following, we prove that the read-write protocols (Algorithm 8 and Algorithm 9) implements atomic storage.

Storage liveness. We have to prove that every read-write operation executed by a correct client eventually terminates. To do so, we prove the following theorem.

Theorem 5. Every read-write operation executed by a correct client eventually terminates.

Proof. Consider a read-write operation o executed by a correct client c. For the first phase phase1 from o, client c sends a message to all servers and waits for servers' responses. There are three cases as follows:

1) A weighted majority of servers that their epochs are equal to $c.cepoch$ responds to client c. Consequently, client c ends phase1 and starts the execution of the second phase phase2 from o by sending a message to all servers and waits for servers' responses. There are three following sub-cases:

   1.1) A weighted majority of servers that their epochs are equal to $c.cepoch$ responds to client c. Consequently, client c ends phase2, and the operation terminates.

   1.2) Client c receives a response from a server with epoch e such that $c.cepoch \neq e$. If $c.cepoch < e$, client c updates its current epoch. Then, the client restarts the operation. We just need to show that this sub-case occurs a finite number of times, i.e., the operation o is not restarted infinitely many times. We can conclude from Lemma 6 that the process of uninstalling epoch *lepoch* eventually terminates. Besides, according to Assumption 4,

Algorithm 8 The reader-writer side of the read-write protocols - process p_i

variables

1) $cnt \leftarrow 0$, $cepoch \leftarrow e_0$

functions to read-write the atomic storage

2) read() $\equiv$ read_write($\perp$)
3) write($value$) $\equiv$ read_write($value$)

function read_write($value$)

phase1

4) $cnt \leftarrow cnt + 1$
5) send $\langle \text{READ}, cnt \rangle$ to all servers
6) $M \leftarrow \varnothing$
7) **repeat**
8)   **upon receipt of** message $\langle \text{READACK}, reg, cnt, e, w \rangle$
9)   **if** $cepoch = e$
10) $M \leftarrow M \cup \{\langle reg, cnt, e, w \rangle\}$
11)   **else**
12)   **if** $cepoch < e$
13) $cepoch \leftarrow e$
14)   read_write($value$) ▷ restart the operation
15) **until** is_quorum(M)
16) $maxtag \leftarrow \text{max}\big(\{m.reg.tag \mid m \in M\}\big)$
17) $maxreg \leftarrow \text{find}\big(\{m.reg \mid m \in M \textbf{ and } m.reg.tag = maxtag\}\big)$
18) **if** $value = \perp$
19) $value \leftarrow maxreg.value$
20) **else**
21) $ts \leftarrow maxtag.ts + 1$
22) $pid \leftarrow p_i$

phase2

23) send $\langle \text{WRITE}, \langle\langle ts, pid \rangle, value \rangle, cnt, e \rangle$ to all servers
24) $M \leftarrow \varnothing$
25) **repeat**
26)   **upon receipt of** message $\langle \text{WRITEACK}, reg, cnt, e, w \rangle$
27)   **if** $cepoch = e$
28) $M \leftarrow M \cup \{\langle reg, cnt, e, w \rangle\}$
29)   **else**
30)   **if** $cepoch < e$
31) $cepoch \leftarrow e$
32)   read_write($value$) ▷ restart the operation
33) **until** is_quorum(M)
34) **return** $value$

the number of epochs are finite. Hence, the operation o will eventually be executed.

1.3) Operation o is concurrent with uninstalling epoch *lepoch* so that read-write operations are disabled. We can conclude from Lemma 6 that the process of uninstalling epoch *lepoch* eventually terminates. After that,

Algorithm 9 The server side of the read-write protocols - server s_i

variables
1) $register[tag[ts, pid], val] \leftarrow \langle\langle 0, \perp\rangle, \perp\rangle$

function tag()
2) **return** $register.tag$

function register()
3) **return** $register$

function set_register(ts, pid, val)
4) **return** $register \leftarrow \langle\langle ts, pid\rangle, val\rangle$

upon receipt of message $\langle$READ$, cnt\rangle$ from process p
5) $cepoch \leftarrow$ cepoch() from Algorithm 7
6) $weight \leftarrow$ weight($cepoch$)
7) **send** $\langle$READACK$, \langle tag, val\rangle, cnt, cepoch, weight\rangle$ to process p

upon receipt of message $\langle$WRITE$, \langle tag, val\rangle, cnt, e\rangle$ from process p
8) $cepoch \leftarrow$ cepoch() from Algorithm 7
9) $weight \leftarrow$ weight($cepoch$)
10) **if** $cepoch = e$
11)   **if** $register.tag < tag$
12) $register \leftarrow \langle tag, val\rangle$
13) **send** $\langle$WRITEACK$, cnt, cepoch, weight\rangle$ to process p

read-write operations are enabled again. Then, we reach to other cases or sub-cases.

2) Client c receives a response from a server with epoch e such that $c.cepoch \neq e$. If $c.cepoch < e$, client c updates its current. Then, the client restarts the operation (cannot start phase phase2). This case is similar to sub-case 1.2.

3) Operation o is concurrent with uninstalling epoch $lepoch$ so that read-write operations are disabled. This case is similar to sub-case 1.3.

□

Storage atomicity. We have to prove that the read-write protocols of our weighted storage implement an atomic read-write register (Definition 5). The sketch of the proof is as follows.

Lemma 7. *Each phase of a read-write operation always finishes in the last installed epoch in the system (lepoch).*

Proof. We prove the lemma by contradiction. For the sake of contradiction, consider a client c that finishes a phase ph by receiving a quorum of replies from servers (say, ρ) in epoch $v < lepoch$. Without loss of generality, assume that $e.succ = lepoch$. According to Lemma 2, by installing epoch $lepoch$ in the system, at least a weighted majority of servers (say, ρ') in epoch e had uninstalled epoch e; therefore, a weighted majority of servers in epoch e cannot

participate in phase *ph*. Since $\rho \cap \rho' = \varnothing$ (quorum intersection property is not satisfied), we have a contradiction. □

Lemma 8. *Each read-write operation always finishes in the last installed epoch in the system (lepoch).*

Proof. Both read and write phases of each read-write operation of Algorithm 8, are executed in the same epoch. Each phase of a read-write operation always finishes in the last installed epoch in the system according to Lemma 7 that completes the proof. □

Let $\mathcal{R}$ be the register and $write(\beta)$ be the operation to write the value β in $\mathcal{R}$ with an associated tag $tag(\beta)$.

Lemma 9. *Let α be the value of last write operation completed in epoch $e =$ lepoch. Then, in the next epoch e.succ, one of the following cases may happen. (1) If there is no concurrent write operation with changing epoch from e to e.succ, then α is propagated to epoch e.succ. (2) If changing epoch from e to e.succ is concurrent with a write operation $write(\beta)$ such that $tag(\alpha) < tag(\beta)$, then either α or β is propagated to e.succ.*

Proof. Case 1. Since there is no concurrent write operation with changing epoch from e to $e.succ$, servers by executing Lines 17-22 of Algorithm 7, propagate α to epoch $e.succ$. *Case 2.* Three following sub-cases should be considered. (a) No server in e updated its local state with $\beta, tag(\beta)$ before installing $e.succ$. We can reduce this case to the Case 1. (b) Some servers in e store $\beta, tag(\beta)$ and some servers store $\alpha, tag(\alpha)$. Due to Lemma 7 and Lemma 8, client restarts $write(\beta)$ and tries it later. (c) A weighted majority of servers in e stores $\beta, tag(\beta)$, so this value should be written to epoch $e.succ$. □

Lemma 10. *Assume that a read operation $read_1$ returns a value α_1 at time t_1^e, which has an associated tag tag_1. A read operation $read_2$ started at time $t_2^s > t_1^e$ returns a value α_2 associated with a tag tag_2 such that either: (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 $read_1$ starts at time t_1^s and $read_2$ ends at time t_2^e, then $t_1^s < t_1^e < t_2^s < t_2^e$. There are four possible, mutually exclusive, following cases.

Case 1. There is no concurrent changing epoch. For this case, the behavior of the algorithm is the same of the basic protocol. *Case 2.* There is a concurrent changing epoch with $read_1$ ($t_1^s < t' < t_1^e$). Since $t' < t_1^e < t_2^s$, the epoch changer algorithm installs $e.succ$ before both s_2 starts and s_1 ends. Consequently, from Lemma 7 and Lemma 8 both $read_1$ and $read_2$ finish in epoch $e.succ$. Now, there is no difference between this case and Case 1. *Case 3.* There is a concurrent changing epoch between $read_1$ and $read_2$ ($t_1^e < t' < t_2^s$). By using Lemma 9, we can ensure the correctness of this lemma. *Case 4.* There is a concurrent changing epoch with $read_2$ ($t_2^s < t' < t_2^e$). This case is similar to Case 2. □

Theorem 6. The read-write protocols implement an atomic read-write register.

Proof. This proof follows directly from Lemma 10. □

4.6 Monitoring System

We focus on considering latency instead of throughput because throughput can be effectively improved by adding more resources (CPU, memory, faster disks) to servers or using better links. However, latency in geo-distributed storage systems will be affected by the speed of light limit and perturbations caused by bandwidth sharing [58].

In the WMQS, a weight is assigned to each server based on the server's latency or throughput.

In the process of assigning weights to the servers, the weights of servers should be in accordance with the latency scores of servers to improve the quorum latency of the storage system. In further detail, if the latency score of server s_i is greater than s_j, then $w_j < w_i$. The first step for assigning such weights to servers is to present a monitoring system by which the latency scores of servers can be computed. The monitoring system is executed by servers and presents the following operations:

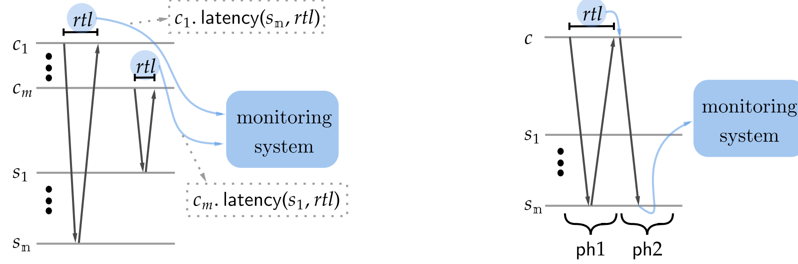
- $\mathsf{latency}(s, rtl)$ that can be called by a client $c \in C$ and states that the round-trip latency of a communication between client c and server $s \in S$ is equal to rtl,
- $\mathsf{lscore}(s)$ that can be called by each server and returns the latency score of server $s \in S$.

Each client calls operation latency to provide input to the monitoring system (Figure 4.3a). Then, the monitoring system computes the latency scores of servers based on such inputs. The monitoring system can be implemented in different ways; however, to implement it efficiently, clients utilize only the messages used to execute read-write operations[1] in such a way that (a) the round-trip latencies of client-server communications are measured using the first phases of read-write operations, (b) such measured latencies are sent to the monitoring system using the messages of the second phases of the operations (Figure 4.3b).

The implementation consists of two modules– ClientServerMS and ServerServerMS. Each server $s_i \in S$ has an ordered set to store the latency scores of servers, denoted by $lscores_i = \langle ls_1, \ldots, ls_j, \ldots, ls_n \rangle$, where ls_j is a variable to store the latency score of server s_j.

ClientServerMS module. Each client c, for the first phase ($ph1$) of each read-write operation, creates an ordered set denoted by $rtls = \langle rtl_1, \ldots, rtl_j, \ldots, rtl_n \rangle$ to store the round-trip latencies of servers for executing $ph1$ (Figure 4.4). In further detail, the value of variable rtl_j is computed as follows: $rtl_j = \mathsf{min}(mrtl, t_j^2 - t_j^1)$, where $mrtl$ is a pre-defined constant for the maximum round-trip latency, t_j^1 is the time that client c sends the READ message to server s_j, and t_j^2 is

[1] As a reminder, each read-write operation executed by client $c \in C$ consists of two consecutive phases– $ph1$ and $ph2$. In phase $ph1$, client c sends a READ message to (a quorum of) servers. Then, client c waits until receiving READACK messages from a quorum of servers. After finishing $ph1$, client c starts $ph2$ by sending a WRITE message to (a quorum of) servers.



(a) The inputs of the monitoring system are the round-trip latencies of client-server communications. Each client calls operation latency to send the round-trip latency of its communication with a server to the monitoring system.

(b) Clients utilize the messages used to execute the phases– $ph1$ and $ph2$– of read-write operations for measuring the round-trip latencies of client-server communications and sending such latencies to the monitoring system.

Figure 4.3: The relationship between the clients and the monitoring system

the time that client c receives the READACK message sent by s_j. Note that a client might finish the first phase of a read-write operation and start the second phase of the operation while it has not received the READACK messages of some servers. In such situations, it is required to use $mrtl$ as the round-trip latency of the communications with the servers that their messages have not been received.

Client c, after finishing the first phase and by starting the second phase of the operation, attaches the ordered set $rtls$ to its WRITE messages (Figure 4.4). Each server s_i has a set called $history_rtls$ to store every $rtls$ that is attached to a received WRITE message. Servers can compute the latency scores of servers based on set $history_rtls$ in different ways. However, we use the following function named history__rtls2lscores to compute the latency scores of servers:

▷ the algorithm executed by server s_i
▷ $lscores_i = \langle ls_1, \ldots, ls_j, \ldots, ls_n \rangle$

function history__rtls2lscores($history_rtls$)
1) $\langle tls_1, \ldots, tls_j, \ldots, tls_n \rangle \leftarrow \langle 0, \ldots, 0, \ldots, 0 \rangle$
2) **for** $j = 1 : n$
3) $r_1 \leftarrow$ sort($\{rtls[j] \mid \forall\ rtls \in history_rtls\}$)
4) $r_2 \leftarrow$ remove the first $1/3$ and the last $1/3$ values of r_1
5) $tls_j \leftarrow$ average(r_2)
6) $lscores_i \leftarrow (lscores_i + \langle tls_1, \ldots, tls_j, \ldots, tls_n \rangle)/2$

Function history__rtls2lscores takes as an input set $history_rtls$. To compute the latency score of server s_j: (a) the round-trip latencies of server s_j stored

in every set $rtls \in history_rtls$ are extracted, then the extracted values are sorted (Line 3), (b) the outlier values are removed from the sorted set (Line 4), (c) the average of the remaining set is computed as the temporary latency score of server s_j (Line 5), and (d) the average of the temporary latency score of server s_j and the stored one in set $lscores_i$ are computed as the latency score of server s_j (Line 6). To remove the outlier values, we simply remove the first 1/3 and the last 1/3 values of the sorted set.

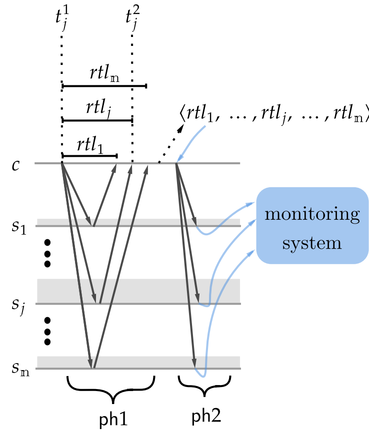


Figure 4.4: ClientServerMS module of the monitoring system

ServerServerMS module. Each server s_i broadcasts set $lscores_i$ frequently. On the other hand, each server s_i, by receiving set $lscores_j$ from server s_j, substitues the average of $lscores_i$ and $lscores_j$ with its set. Having the ServerServerMS module in the implementation of the monitoring system is necessary because a server s_i might not receive a WRITE message from a client while another server s_j can receive; servers s_i can retrieve the unreceived information using the ServerServerMS module.

4.7 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) MW-ABD [8] that uses an MQS, and (2) RAMBO [25] (a consensus-based reconfiguration protocol). We implemented prototypes of the MW-ABD, RAMBO, and our protocols in the python programming language. Besides, we used KOLLAPS [26], a fully distributed network emulator, to create the network and links' latencies.

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 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 ($0 < \delta$ is a constant) as follows to increase/decrease its weight. Assume that the latency score of server s are ls_t and $ls_{t'}$ respectively at time t and $t' = t+\delta$. Also, assume that the total latency scores of servers computed by s are LS_t and $LS_{t'}$ respectively at time t and t'. Server s calls function *increase* (resp. *decrease*) to increase (resp. decrease) its weight at time t' if $ls_t/LS_t + \tau < ls_{t'}/LS_{t'}$ (resp. $ls_{t'}/LS_{t'} + \tau < ls_t/LS_t$), where τ 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 [46], and the containers communicate through the Docker Swarm [1]. 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 read-write request as soon as receiving the response of the previously sent read-write request. Since there is no difference between read-write protocols regarding the number of communication rounds in our read-write protocols, we set the read-write 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 Δ. We executed 100 runs and computed the average of the results that is depicted in Figure 4.5. The average quorum latencies of the MW-ABD, RAMBO, and our protocol are 139, 118, and 101 milliseconds, respectively. The MW-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 MW-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.

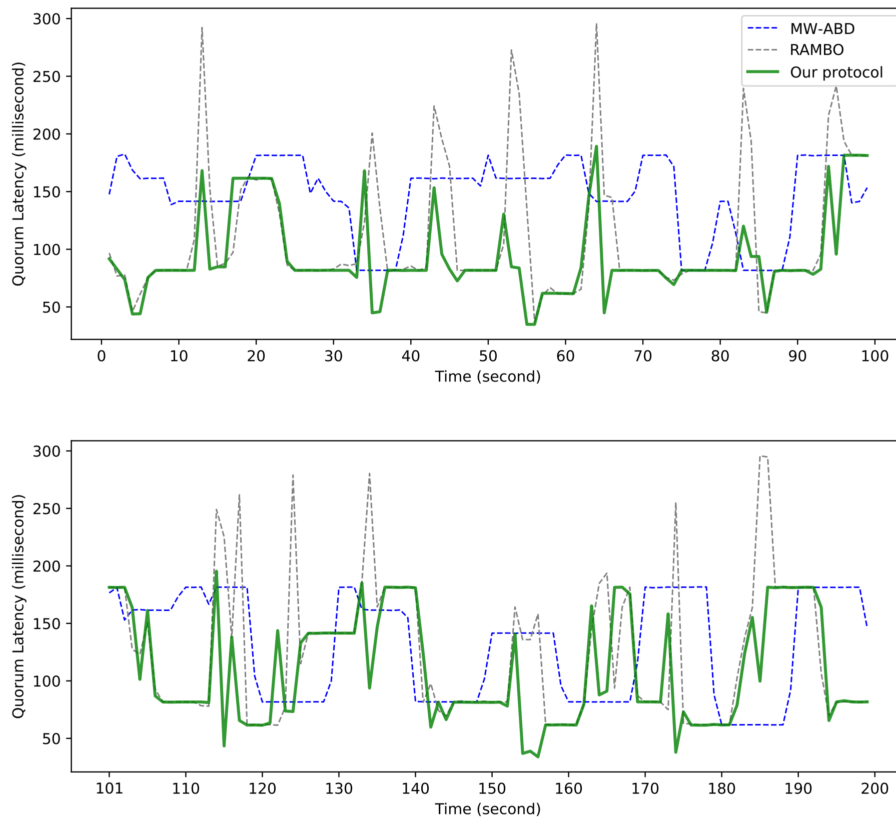


Figure 4.5: Quorum latency evaluation for our protocol, MW-ABD, and RAMBO.

Chapter 5

Epoch-Less Weight Reassignment

Abstract. In the epoch-based weight reassignment protocol presented in Chapter 4, donations and retakes performed during an epoch e cannot be applied unless e is changed to $e + 1$, and all correct servers should participate in the process of synchronizing states. Besides, read-write operations might be restarted when the epochs are changed to guarantee the liveness property of the atomic storage. This chapter presents an epoch-less weight reassignment protocol for implementing the WeakPairwise approach, in which each donation and retake can be applied without requiring epochs. Consequently, donations and retakes can be applied faster, and it is not required to restart read-write operations. A few optimization techniques are presented to improve the performance of the epoch-less protocol further. One of the optimization techniques is to reduce the number of participants in the process of synchronizing states to only two servers.

5.1 Warmup

In this section, we present the overall idea of the epoch-less implementation of the WeakPairwise abstraction. Assume that server s_i donates some part of its weight, that is equal to a constant w, to another server s_j. In the epoch-based protocol, the donation process is started by sending a DNT message from s_i to s_j, that contains w and a sequence number dedicated to the donation. Server s_i (resp. s_j) should wait until changing the current epoch e_k to the next epoch e_{k+1} for decreasing (resp. increasing) its weight by w, and a quorum of correct servers should participate in the process of synchronizing the states. Then, server s_i can retake its donated weight at the time changing epoch e_{k+1} to e_{k+2}.

However, in the epoch-less protocol, server s_i (resp. s_j) creates a change to decrease (resp. increase) its weight by w (resp. $w' \leq w$) as soon as sending (resp. receiving) such a message. Then, server s_i starts a timeout called τ_{retake} for the

donation. After finishing such a timeout, server s_i asks a quorum of servers Q to assist it to retake its donated weight by broadcasting a RTK message. Each server $s \in Q$ creates an auxiliary change (*achange*) and adds the created *achange* to another set called *achanges*. An *achange* is a similar data structure to change that is created by a server to assist a donor that wants to retake its donated weight. In case every server $s \in Q$ creates an *achange* and the donee is correct, the donee cannot use the donated weight anymore. Each server $s \in Q$ sends a confirmation to the donor within a RTN message after creating an *achange*. If the donor receives confirmations from all servers Q, it creates a change to increase its weight. Every change is added to set *chagnes*. The whole process is depicted in Figure 5.1.

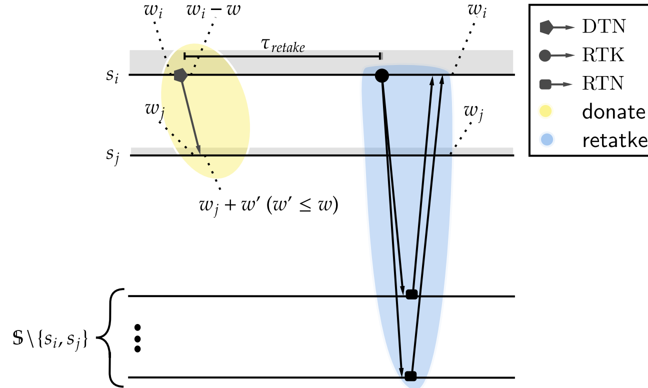


Figure 5.1: Ilustration of donate and retake operations in the epoch-less weight reassignment protocol

In this chapter, we explain the epoch-less protocol in further details. First, we explain the donate operation in Section 5.2. Particularly, we determine (a) the conditions that each donor should satisfy to be allowed to start a donation, (b) what a donee must do when it receives the donated weight. Then, we present the retake process in Section 5.3. A few optimization techniques are presented in Section 5.4 to improve the performance of the epoch-less weight reassignment protocol. We present the read-write protocols and an example that contains a few donations, retakes, and clients' read-write requests in Section 5.5.

5.2 Donate Operation

Algorithm 10 is the pseudocode of the donate operation. Each server has a variable called *sn* initialized with 0 and used as a sequence number to distinguish its donations. Also, each server has a set denoted by *changes* to store every received change initialized with a change $\langle iw, s_i, - \rangle$ and a variable denoted by *register* to store the tag and value of its local register. Each server has access to two functions: • donated_but_not_retaken that determines the amount of

weight that is donated by a server but not retaken, and • read that returns the tag and value of the atomic storage. The implementations of these functions are presented later in this section.

Each server s_i can donate some part of its weight equal to w to another server s_j if the following conditions are satisfied:

C1) Each server s_i has a timeout called $\tau_{donation}$; if the timeout finishes, server s_i can start a donation. Such a timeout is required to control the number of donations made by each server.

C2) Server s_i should have some weight to donate to s_j. Each server s_i is allowed to donate at most $iw - minw$ part of its weight. Indeed, server s_i is allowed to donate weight w to another server s_j if the following relation holds: $iw - minw \leq \mathsf{donated_but_not_retaken}(s_i) - w$.

C3) Server s_i's latency score should be significantly greater than s_j's latency score, i.e., $\mathsf{lscore}(s_j) \ll \mathsf{lscore}(s_i)$. Note that the reason for not using 'greater than' and using 'significantly greater than' is to avoid unnecessary weight reassignments.

If the conditions mentioned above are met, server s_i takes the following steps: (a) creates a change for the donation and adds it to set *changes*, (b) restarts timeout $\tau_{donation}$, (c) starts a timeout called τ_{retake} for the donation, and (d) donates w to s_j by sending a DNT message (Lines 4-10 of Algorithm 10). According to the following theorem and its corollary, decreasing the weight of a server does not create a safety issue for atomic storage.

Theorem 7. Assume that the following relation holds for server s_i: $iw - minw + w \leq \mathsf{donated_but_not_retaken}(s_i)$. If s_i's weight is decreased by w while tw is not changed, the safety property of atomic storage is not violated, even without updating the servers' local registers.

Proof of Theorem 7. To prove this theorem, we present all possible cases. Then, we show that no case violates the safety property. To do so, assume that the last completed write operation was $\langle tag_\beta, \beta\rangle$. One of the following disjoint cases can happen:

1) The state of s_i is $\langle tag_\beta, \beta\rangle$, and there is no concurrent write at the time of changing s_i's weight. Assume that servers are grouped based on their tags in such a way that group G_t contains all servers with tag t. Accordingly, $s_i \in G_{tag_\beta}$. By decreasing s_i's weight, just the total weight of group G_{tag_β} will be decreased. Note that the total weight of each group $G_t \neq G_{tag_\beta}$ is less than $tw/2$ because the last write was done by servers of group G_{tag_β}, and tw is not changed. On the other hand, other groups' total weights will not change by decreasing s_i's weight. Since there is no group $G_t \neq G_{tag_\beta}$ with a total weight greater than $tw/2$, it is not possible to read a stale value.

2) The state of s_i is $\langle tag_\beta, \beta\rangle$, and there is at least one concurrent write at the time of changing s_i's weight. Similar to Case 1, it is not possible to read a stale value.

3) The state of s_i is $\langle tag_\alpha, \alpha \rangle$, where $tag_\alpha < tag_\beta$, and there is no concurrent write at the time of changing s_i's weight. It is clear that there is no need to synchronize the states because the states of at least a weighted majority of servers are equal to $\langle tag_\beta, \beta \rangle$, and by decreasing s_i's weight, nothing can happen for that weighted majority.

4) The state of s_i is $\langle tag_\alpha, \alpha \rangle$, where $tag_\alpha < tag_\beta$, and there is at least one concurrent write at the time of changing s_i's weight. Similar to Cases 2 and 3, it is not possible to read a stale value.

5) The state of s_i is $\langle tag_\gamma, \gamma \rangle$, where $tag(\beta) < tag(\gamma)$, and there is no concurrent write at the time of changing s_i's weight. This case can happen if the write $\langle tag_\gamma, \gamma \rangle$ has not completed at the time of decreasing s_i's weight. Similar to Case 1, it is not possible to read a stale value.

6) The state of s_i is $\langle tag_\gamma, \gamma \rangle$, where $tag(\beta) < tag(\gamma)$, and there is at least one concurrent write at the time of changing s_i's weight. This case can happen if the write $\langle tag_\gamma, \gamma \rangle$ has not completed at the time of decreasing s_i's weight. Similar to Case 1, it is not possible to read a stale value.

None of cases mentioned above violet the safety property. Moreover, since there is no other case, the theorem holds. □

Corollary 2. According to Theorem 7, during a donation made from a server s_i to another server s_j, the safety property of atomic storage is not violated if the weight of server s_i decreases by w and the servers' local registers are not updated.

After receiving a DNT message from s_j, server s_i accepts the donated weight w if the following condition is satisfied:

C4) Server s_i's latency score should be significantly less than s_j's latency score, i.e., $\mathsf{lscore}(s_i) \ll \mathsf{lscore}(s_j)$.

If the condition mentioned above is met, server s_i creates a change for the received donation and adds the change to set *changes* so that its weight increases by $\mathsf{min}(maxw - w_i, w)$. Theorem 8 states that it is required to update the state of server s_i before doing such a weight increment (Lines 11-13 of Algorithm 10).

Theorem 8. Assume that a server s_i with tag tag_i receives either a DNT message from another server s_j or RTN messages from a quorum of servers. Consequently, its weight might be increased. The safety property of atomic storage might be violated if the local register of server s_i is not updated before increasing its weight.

Proof of Theorem 8. This theorem is proved by contradiction. For the sake of contradiction, assume that the safety property is not violated if the state of server s_i is not synchronized before increasing its weight. Consider the following example that is depicted in Figure 5.2. Let $S = \{s_1, s_2, s_3, s_4\}$, $f = 1$, and

$\Delta = 1$. Accordingly, $tw = 5$ and $iw = 1.25$. Assume that, at time t_1, the weights of servers s_1, s_2, s_3, and s_4 are 2, 1, 1, and 1, respectively. Also, assume that the last written value is α with tag tag_α and is stored by servers s_1 and s_2.

Server s_1 returns the donated weights to other servers at time t_2. Decreasing s_1's weight does not create a safety issue according to Theorem 7. When servers s_2, s_3, and s_4 receive the returned weights, their weights are increased but there is no safety violation if the states are not synchronized due to the assumption made in the beginning of the proof. Then, each server $s \in \{s_1, s_2, s_3\}$ donates weight $w = 0.25$ to server s_4. Decreasing the weight of server $s \in \{s_1, s_2, s_3\}$ weight does not create a safety issue according to Theorem 7. Consequently, the new weights of servers s_1, s_2, s_3, and s_4 are 1, 1, 1, and 2, respectively. Note that the weight of server s_4 can be increased without requiring to synchronize the states.

At time t_3, client c sends a read request to servers s_3 and s_4. A quorum is constituted by servers s_3 and s_4 because their total weight is greater than $tw/2$, and result of the read request is $\perp$ while it is not the last written value. We find a safety violation, that is a contradiction as well so the theorem holds.

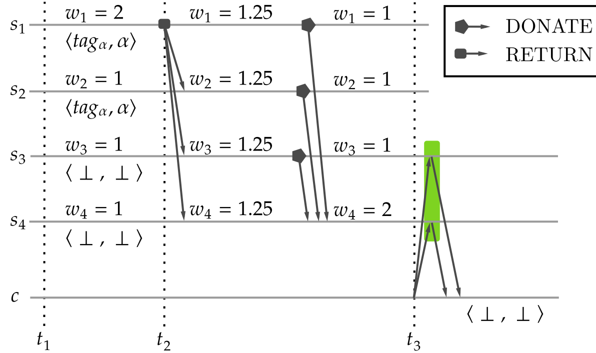


Figure 5.2: An example of violating the safety property

□

5.3 Retake Operation

Algorithm 11 is the pseudocode of the retake operation. Each server has a set denoted by *achanges* to store auxiliary changes and has access to function is_quorum. Function is_quorum determines whether a subset of servers is a quorum. This operation begins if a server s_i has donated some part of its weight to another server s_j, and after a while, s_i wants to retake the donated weight. In further detail, after sending the DNT message, server s_i starts a timeout τ_{retake} for the donation. As soon as finishing such a timeout, server s_i

Algorithm 10 The donate operation - server s_i

variables

1) $sn \leftarrow 0$
2) $changes \leftarrow \{\langle iw, s_i, - \rangle\}$
3) $register[tag[ts, pid], val] \leftarrow \langle\langle 0, \perp\rangle, \perp\rangle$

upon timeout for $\tau_{donation}$

4) **if** $iw - minw \leq$ donated_but_not_retaken$(s_i) - w$
   and $\exists\ s_j \neq s_i$: lscore$(s_j) \ll$ lscore(s_i)
5) $sn \leftarrow sn + 1$
6) $d \leftarrow \langle \text{DNT}, s_i, sn, w, s_j \rangle$
7) $changes \leftarrow changes \cup \{\langle -w, s_i, d\rangle\}$
8) **restart** timeout $\tau_{donation}$
9) **start** timeout τ_{retake} for d
10) **send** d to s_j

upon receipt of $d = \langle \text{DNT}, s_j, s, w, s_i \rangle$ from s_j

11) **if** lscore$(s_i) \ll$ lscore(s_j)
12) $register \leftarrow$ read()
13) $changes \leftarrow changes \cup \{\langle w, s_i, d\rangle\}$

broadcasts a RTK message to all servers using a **RB_broadcast** module (Line 2 of Algorithm 11).

Upon delivering a RTK message from **RB_deliver**, each server s_k creates an auxiliary change $\langle -w, s_j, \langle \text{RTK}, s_i, sn, w, s_j\rangle\rangle$ for the donation. Then, the created *achange* is added to set *achanges*, and a RTN message is sent to s_i (Lines 3-4 of Algorithm 11). Each server includes its *changes* and *achanges* to the sending RTN message to s_i. Upon receiving a RTN message from a quorum of servers, server s_i executes a read operation to avoid creating a safety issue due to Theorem 8. Then, it creates a change and adds the created change to its *changes*. (Lines 5-6 of Algorithm 11). Server s_i utilizes the information included in the RTN messages to decide whether a quorum is constituted using function is_quorum.

Consider a process p that receives messages from a subset of servers $S \subseteq S$ such that each server $s \in S$ includes its *changes* and *achanges* in its message. By using function is_quorum, that takes as input set S, process p can determine whether a quorum is constituted. Function is_quorum is based on function weight. Function weight takes as its input a server s_i and set S. Then, such a function calculates and returns the weight of server s_i as follows: if $s_i \notin S$, zero is returned as the weight of s_i; otherwise, the summation of three following values are returned: • iw, • the summation of all weights donated, received, and retaken by s_i, and • the summation of all weights that are donated to s_i such that their donors sent RTK messages using the **RB_broadcast** module (Line 2 of Algorithm 11) but s_i have not received such messages yet, i.e., the summation of all *achanges* that are stored by servers of S about s_i but it is not stored by

Algorithm 11 The retake operation - server s_i

variables
1) $achanges \leftarrow \varnothing$

upon timeout τ_{return} for $\langle \text{DNT}, s_i, s, w, s_j \rangle$
2) **RB_broadcast** $\langle \text{RTK}, s_i, s, w, s_j \rangle$

upon RB_deliver $\langle \text{RTK}, s_j, s, w, s_k \rangle$
3) $achanges \leftarrow achanges \cup \{\langle -w, s_k, \langle \text{RTK}, s_j, s, w, s_k \rangle\rangle\}$
4) **send** $\langle \text{RTN}, s, changes, achanges \rangle$ to s_j

upon receipt of $\langle \text{RTN}, s, *, * \rangle$ from a quorum
5) $register \leftarrow \mathsf{read}()$
6) $changes \leftarrow changes \ \cup \{\langle w, s_i, \langle \text{RTK}, s_j, s, w, s_i \rangle\rangle\}$

function donated_but_not_retaken(s_i)
7) **return** $\mathsf{sum}\big(\{w \mid \langle w, s_i, \langle \text{DNT}, *, s, *, * \rangle\rangle \in changes$
$\langle w, s_i, \langle \text{RTK}, *, s, *, * \rangle\rangle \notin changes\}\big)$

s_i.

function weight(s_i, S)
1) **if** $s_i \notin S$
2) **return** 0
3) **else**
4) $w \leftarrow \mathsf{sum}\big(\{w \mid \langle w, s_i, * \rangle \in s_i.changes\}\big) +$
$\mathsf{sum}\big(\{w \mid \langle w, s_i, \langle \text{RTK}, *, sn, *, * \rangle\rangle \in s_*.achanges$
and $\langle w, s_i, \langle \text{RTK}, *, sn, *, * \rangle\rangle \notin s_i.changes\}\big)$
5) **return** $\mathsf{min}(w, maxw)$

In function is_quorum, in case the total weight of servers S is greater than $tw/2$, S is a quorum.

function is_quorum(S)
1) **if** $tw/2 < sum\big(\{\mathsf{weight}(s_i, S) \mid s_i \in S\}\big)$
2) **return** *yes*
3) **else**
4) **return** *no*

5.4 Optimization

This section presents a few optimization techniques that can be used to improve the performance of the epoch-less weight reassignment protocol.

Technique 1

Given a donation made from server s_i to server s_j, if server s_j's weight becomes greater than $maxw$ by adding the donated weight, server s_j cannot use the donated weight. Besides, server s_i cannot use the donated weight until finishing the timeout dedicated to the donation. To improve the efficiency of this process, server s_j sends a message to server s_i by which the donated weight is returned to server s_i. Server s_i uses the donated weight as soon as receiving the returned message.

Technique 2

Each server asks all servers for retaking its donated weights To improve the efficiency of the retake process, each server can batch its retake requests if it has multiple retake requests that should be broadcasted to all servers. Then, all the bached requests can be sent in a single request.

Technique 3

For each donation, the donor of the donation broadcasts a RTK message after finishing the timeout τ_{retake} devoted to the donation. If the latency scores of the donor and donee have not changed from the time of sending the DNT message to finishing the timeout τ_{retake} devoted to the donation, it is not required to retake the donated weight. Indeed, the donor can extend the lifetime of the donation. Accordingly, the donor can only reset its timeout τ_{retake} devoted to the donation without sending a RTK message.

5.5 Read-Write Protocols

Here, we present the read-write protocols of the epoch-less weight reassignment protocol.

Reader-writer side. Algorithm 12 is the pseudocode of the reader-writers' algorithm. The reader-writers' algorithm is similar to the reader-writer's algorithm of the MW-ABD protocol [45] with only one difference: each reader or writer, after receiving messages from a set $S' \subseteq S$ to decide whether a quorum is constituted, calls function is_quorum (Lines 10 and 23 of Algorithm 12).

Server side. Algorithm 13 is the pseudocode of the servers' algorithm. The servers' algorithm is similar to the servers' algorithm of the MW-ABD protocol with only one difference: each server includes its *changes* and *achanges* to its responses.

Correctness. The following theorems states an atomic storage can be implemented using the read-write protocols (Algorithm 12 and Algorithm 13) if the weights of servers are reassigned by calling donate (Algorithm 10) and retake (Algorithm 11) operations.

Algorithm 12 The reader-writer side of the read-write protocols - process p_i

variables
1) $opCnt \leftarrow 0$

functions for read-write atomic storage
2) $\mathsf{read}() \equiv \mathsf{read_write}(\bot)$
3) $\mathsf{write}(value) \equiv \mathsf{read_write}(value)$

function $\mathsf{read_write}(value)$
ph1
4) $opCnt \leftarrow opCnt + 1$
5) **send** $\langle \text{READ}, opCnt \rangle$ to all servers
6) $S \leftarrow \varnothing$
7) **repeat**
8) **upon receipt of** $\langle \text{READACK}, reg, opCnt, changes, achanges \rangle$ from s_i
9) $S \leftarrow S \cup s_i.\langle reg, changes, achanges \rangle$
10) **until** $\mathsf{is_quorum}(S)$
11) $maxtag \leftarrow \mathsf{max}\big(\{s_i.reg.tag \mid s_i \in S\}\big)$
12) $maxreg \leftarrow \mathsf{find}\big(\{s_i.reg \mid s_i \in S$ **and** $s_i.reg.tag = maxtag\}\big)$
13) **if** $value = \bot$
14) $value \leftarrow maxreg.value$
15) **else**
16) $ts \leftarrow maxtag.ts + 1$
17) $pid \leftarrow p_i$
ph2
18) **send** $\langle \text{WRITE}, \langle \langle ts, pid \rangle, value \rangle, opCnt \rangle$ to all servers
19) $S \leftarrow \varnothing$
20) **repeat**
21) **upon receipt of** $\langle \text{WRITEACK}, reg, opCnt, changes, achanges \rangle$ from s_i
22) $S \leftarrow S \cup s_i.\langle reg, changes, achanges \rangle$
23) **until** $\mathsf{is_quorum}(S)$
24) **return** $value$

Safety of the atomic storage

Lemma 11. *Assume that there is no change from time t_1 to time t_2. Also, assume that set $S_1 \subseteq S$ (resp. $S_2 \subseteq S$) 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, $S_1 \cap S_2 \neq \varnothing$.*

Proof. A set $S' \subseteq S$ is determined as a quorum by function is_quorum if the

Algorithm 13 The server side of the read-write protocols - server s_i

upon receipt of $\langle$READ, $cnt\rangle$ from p
1) **send** $\langle$READACK, $register, cnt, changes,$
   $achanges\rangle$ to p

upon receipt of $\langle$WRITE, $\langle tag, val\rangle, cnt\rangle$ from p
2) **if** $register.tag < tag$
3) $register \leftarrow \langle tag, val\rangle$
4) **send** $\langle$WRITEACK, $cnt, changes, achanges\rangle$ to p

following condition is satisfied: the total weight of servers S' is greater than $tw/2$. We use proof by contradiction to prove $S_1 \cap S_2 \neq \varnothing$, and for the sake of contradiction, we assume that $S_1 \cap S_2 = \varnothing$.

Let tw_1 and tw_2 be equal to the total weight of servers S_1 and S_2, respectively. We have:

$$\begin{cases} tw/2 < tw_1 \\ tw/2 < tw_2 \end{cases} \Rightarrow tw < tw_1 + tw_2$$

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

□

Lemma 12. *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 only one change 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 *change* increases the weight of a server s_i. Since there is only one change, the weights of other servers are not reassigned. Reassigning server s_i's weight is the only factor that causes the constitution of new quorums; since before accomplishing such a weight reassignment, server s_i executes a read operation to update its register, this lemma is similar to Lemma 11. □

Lemma 13. *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 change 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 there are two changes c_1 and c_2 from time t^e_α to time t^s_β. If one of these changes starts after finishing the other one, the lemma holds because the second change executes a read operation.

In case these changes are concurrent, (without loss of generality, assume that changes c_1 and c_2 are made on servers s_1 and s_2, respectively) we have to show that the following quorums have an intersection: (a) a quorum constituted by s_1 (b) a quorum constituted by s_2 (c) a quorum constituted by both s_1 and s_2 (d) a quorum constituted by none of s_1 and s_2. It is straightforward to show that these quorums have an intersection. Consequently, operation r_2 returns the most up-to-date value. □

Lemma 14. *Assume that a* read *operation $read_1$ returns a value α_1 at time t_1^e with an associated tag tag_1. Also, assume that another* read *operation $read_2$ started at time $t_2^s > t_1^e$ returns a value α_2 associated with tag tag_2. Then, one of the following cases happen: (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 $read_1$ starts at time t_1^s and $read_2$ ends at time t_2^e, then $t_1^s < t_1^e < t_2^s < t_2^e$. There are four mutually exclusive cases:

a) There is no concurrent *change*. For this case, the lemma holds according to Lemma 11.

b) There is a concurrent change with $read_1$ at time t' such that $t_1^s < t' < t_1^e$. For this case, the lemma holds according to Lemma 12.

c) There is at least one change between $read_1$ and $read_2$. For this case, the lemma holds according to Lemma 13.

d) There is a concurrent change with $read_2$ at time t' such that $t_2^s < t' < t_2^e$. This case is similar to Case 2.

□

Theorem 9. The storage system implemented using the read-write protocols (Algorithm 12 and Algorithm 13) is an atomic storage.

Proof. According to Lemma 14, the theorem holds. □

Theorem 10. Atomic storage implemented using the read-write protocols (Algorithm 12 and Algorithm 13) is live.

Proof. 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 equal to $n - f$. Since $tw/2 < n - f$, a quorum can be constituted, i.e., the system remains live even in the worst case scenario. □

An example

The following example contains a few calls of donate, retake, and clients' read-write operations (Figure 5.3). Let $S = \{s_1, s_2, s_3, s_4\}$, $C = \{c_1, c_2\}$, $n = 4$, $f = 1$. Accordingly, $iw = 1.25$, $tw = 5$, and $maxw = 2$. The local register of each server is initialized with $\langle \perp, \perp \rangle$. Also, for each server, sets

changes and *achanges* are initialized with $\{\langle iw, s_i, -\rangle\}$ and $\varnothing$, respectively. Client c_1 executes write(α), and registers of servers s_2, s_3, and s_4 are updated. Note that at least three severs are required to constitute a quorum at this time. Server s_1 donates weight 0.25 to s_2. Consequently, $s_1.changes = \{\langle iw, s_1, -\rangle, \langle -0.25, s_1, \langle \text{DNT}, s_1, 1, 0.25, s_2\rangle\rangle\}$, and $w_1 = 1$. On the other hand, server s_2 executes a read operation after receiving the donation. The read operation is executed by quorum $\{s_1, s_2, s_3\}$. Then,
$s_2.changes = \{\langle iw, s_1, -\rangle, \langle +0.25, s_2, \langle \text{DNT}, s_1, 1, 0.25, s_2\rangle\rangle\}$, and $w_2 = 1.5$.

Client c_1 executes write(β). Note that either $\{s_2, s_3\}$ or $\{s_2, s_4\}$ is a quorum at this time. Assume that the constituted quorum is $\{s_2, s_3\}$, and registers of servers s_2 and s_3 are updated. Then, server s_1 sends a RTK message to s_2 for retaking its donated weight. Server s_2 creates a change $\langle -0.25, s_2, \langle \text{RTK}, s_1, 1, 0.25, s_2\rangle\rangle$ and adds the created change to set $s_2.chagnes$ so $w_2 = 1.25$; then, it sends a RTN message to s_1. After receiving the RTN message, server s_1 executes a read operation. The read operation is executed by quorum $\{s_1, s_2, s_3\}$. Then, server s_1 creates a change $\langle 0.25, s_1, \langle \text{RTK}, s_1, 1, 0.25, s_2\rangle\rangle$ and adds the created change to set $s_1.chagnes$ so $w_1 = 1.25$.

Server s_1 donates weight 0.25 to s_3 after a while. Client c_1 executes write(γ), and a quorum by server s_1, s_2, and s_4 is constituted. After that, server s_1 sends a RTK message to s_3 for retaking its donated weight, but server s_3 does not respond the retake request on time so that server s_1 broadcasts a RTK message to all servers. Servers s_2 and s_4 send their responses to s_1; accordingly, server s_1's weight becomes 1.25 again. Also, for each server $s_i \in \{s_1, s_2, s_4\}$, $s_i.achangess = \{\langle -0.25, s_3, \langle \text{RTK}, s_1, 2, 0.25, s_3\rangle\rangle\}$

Client c_1 executes operation read() and a quorum by servers $\{s_2, s_3, s_4\}$ is constituted. Since servers s_2 and s_4 have auxiliary changes to show that the donated weight by server s_1 to s_3 was retaken, change $\langle 0.25, s_3, \langle \text{DNT}, s_1, 2, 0.25, s_3\rangle\rangle \in s_3.changes$ is not considered in computing s_3's weight.

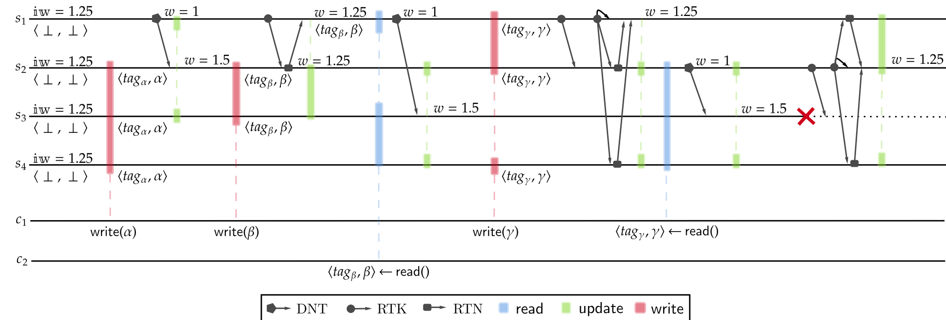


Figure 5.3: An example of calling donate, retake, and clients' read-write operations

Chapter 6

Conclusion

This thesis investigates the problem of enhancing the performance of distributed storage systems based on the majority quorum system and can be implemented in asynchronous, failure-prone distributed systems. We consider a conventional technique to enhance the performance of such systems that is to replace the majority quorum system with the dynamic weighted majority quorum system. Since the dynamic weighted majority quorum system is based on weight reassignment protocols, we formulate the problem of reassigning weights in asynchronous, failure-prone distributed systems.

We investigate three approaches by which weight reassignment protocols can be constructed. We show that two of the approaches cannot be implemented in asynchronous, failure-prone distributed systems regardless of how they are implemented. Then, we conclude that a weight reassignment protocol cannot rely on any weight reassignment approach.

To show the impossibility of implementing a weight reassignment approach in asynchronous, failure-prone distributed systems, we use the notion of *consensus number*. In further detail, we treat a weight reassignment approach as a concurrent object; then, we show how the consensus can be solved using the object; since consensus cannot be implemented in asynchronous, failure-prone distributed systems, we conclude that the weight reassignment approach cannot be implemented in asynchronous, failure-prone distributed systems as well.

We show that one of the investigated approaches can be employed to construct a weight reassignment protocol that can be implemented in asynchronous, failure-prone distributed systems. We propose two weight reassignment protocols based on that approach. Then, we construct atomic storage based on the dynamic weighted majority quorum system such that the servers' weights can be reassigned over time using the proposed weight reassignment protocols.

Due to the similarity of the weight reassignment and reconfiguration problems, we also investigate the reconfiguration problems. We present a new variant of reconfiguration problems called the *conditional reconfiguration* problem. The *conditional reconfiguration* problem is similar to the standard asynchronous reconfiguration problem with only one difference: a condition related to the size of

configurations should be satisfied. We show that the *conditional reconfiguration* problem has no asynchronous implementation.

During the course of this research, we have found several ideas, how the insights and techniques from this thesis could be applied and extended. In this manuscript, 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. We assume that the set of servers does not change over time; however, our work can be extended to consider that servers can leave and new servers can join the system. 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 also be another direction for future work.

Bibliography

[1] Docker swarm. Accessed: 19-10-2021.

[2] Etcd. `https://github.com/etcd-io/etcd`. Accessed: 2022-01-05.

[3] Divyakant Agrawal and Amr El Abbadi. The tree quorum protocol: An efficient approach for managing replicated data. In *VLDB*, volume 90, pages 243–254, 1990.

[4] Marcos K Aguilera, Idit Keidar, Dahlia Malkhi, Jean-Philippe Martin, Alexander Shraer, et al. Reconfiguring replicated atomic storage: A tutorial. *Bulletin of the EATCS*, pages 84–108, 2010.

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

[6] Eduardo Alchieri, Alysson Bessani, Fabíola Greve, and Joni Fraga. Efficient and modular consensus-free reconfiguration for fault-tolerant storage. In *OPODIS*, pages 1–26, 2017.

[7] Yair Amir and Avishai Wool. Optimal availability quorum systems: Theory and practice. *Inf. Process. Lett.*, 65:223–228, 1998.

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

[9] Hagit Attiya, Maurice Herlihy, and Ophir Rachman. Atomic snapshots using lattice agreement. *Distributed Computing*, 8(3):121–132, 1995.

[10] D. Barbara and H. Garcia-Molina. The Reliability of Voting Mechanisms. *IEEE Trans. Comput.*, 36:1197–1208, 1987.

[11] Christian Berger, Hans P. Reiser, Joao Sousa, and Alysson Neves Bessani. Aware: Adaptive wide-area replication for fast and resilient byzantine consensus. *IEEE Transactions on Dependable and Secure Computing*, 2020.

[12] Eric Brewer. Cap twelve years later: How the" rules" have changed. *Computer*, 45:23–29, 2012.

[13] Shun Yan Cheung, Mostafa H Ammar, and Mustaque Ahamad. The grid protocol: A high performance scheme for maintaining replicated data. *IEEE Transactions on Knowledge and Data Engineering*, 4:582–592, 1992.

[14] Gregory Chockler, Rachid Guerraoui, Idit Keidar, and Marko Vukolic. Reliable distributed storage. *Computer*, 42(4):60–67, 2009.

[15] Danco Davcev. A dynamic voting scheme in distributed systems. *IEEE transactions on Software Engineering*, 15:93–97, 1989.

[16] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34:77–97, 1987.

[17] Partha Dutta, Rachid Guerraoui, Ron R Levy, and Arindam Chakraborty. How fast can a distributed atomic read be? In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 236–245, 2004.

[18] Burkhard Englert, Chryssis Georgiou, Peter M Musial, Nicolas Nicolaou, and Alexander A Shvartsman. On the efficiency of atomic multi-reader, multi-writer distributed memory. In *International Conference On Principles Of Distributed Systems*, pages 240–254. Springer, 2009.

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

[20] Eli Gafni and Dahlia Malkhi. Elastic configuration maintenance via a parsimonious speculating snapshot. In *DISC*, pages 140–153, 2015.

[21] C. Georgiou, P. M. Musia, and A. A. Shvartsman. Developing a Consistent Domain-Oriented Distributed Object Service. In *Fourth IEEE International Symposium on Network Computing and Applications*, pages 149–158. 2005.

[22] Chryssis Georgiou, Nicolas Nicolaou, Alexander C Russell, and Alexander A Shvartsman. Towards feasible implementations of low-latency multi-writer atomic registers. In *2011 IEEE 10th International Symposium on Network Computing and Applications*, pages 75–82. IEEE, 2011.

[23] David K. Gifford. Weighted voting for replicated data. *SOSP '79*, pages 150–162, 1979.

[24] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *Acm Sigact News*, 33:51–59, 2002.

[25] Seth Gilbert, Nancy A. Lynch, and Alexander A. Shvartsman. Rambo: a robust, reconfigurable atomic memory service for dynamic networks. *Distrib. Comput.*, 23:225–272, 2010.

[26] Paulo Gouveia, João Neves, Carlos Segarra, Luca Liechti, Shady Issa, Valerio Schiavoni, and Miguel Matos. Kollaps: decentralized and dynamic topology emulation. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[27] Vincent Gramoli, Peter M. Musial, and Alexander A. Shvartsman. Operation Liveness and Gossip Management in a Dynamic Distributed Atomic Data Service. In *PDCS*, pages 206–211, 2005.

[28] Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovič, and Dragos-Adrian Seredinschi. The consensus number of a cryptocurrency. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 307–316, 2019.

[29] Rachid Guerraoui and Luís Rodrigues. *Introduction to Reliable Distributed Programming*. Springer-Verlag, 2006.

[30] Theophanis Hadjistasi, Nicolas Nicolaou, and Alexander A Schwarzmann. Oh-ram! one and a half round atomic memory. In *International Conference on Networked Systems*, pages 117–132. Springer, 2017.

[31] Theophanis Hadjistasi and Alexander A Schwarzmann. Consistent distributed memory services: Resilience and efficiency. In *45th International Colloquium on Automata, Languages, and Programming (ICALP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[32] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13:124–149, 1991.

[33] Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear. *The art of multiprocessor programming*. Newnes, 2020.

[34] Hasan Heydari, Guthemberg Silvestre, and Luciana Arantes. Efficient consensus-free weight reassignment for atomic storage. *https://arXiv.org/abs/2110.10666*.

[35] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, volume 8, 2010.

[36] Sushil Jajodia and David Mutchler. Dynamic voting algorithms for maintaining the consistency of a replicated database. *ACM Transactions on Database Systems*, 15:230–280, 1990.

[37] Leander Jehl and Hein Meling. The case for reconfiguration without consensus: Comparing algorithms for atomic storage. In *OPODIS*, 2017.

[38] Leander Jehl, Roman Vitenberg, and Hein Meling. Smartmerge: A new approach to reconfiguration for atomic storage. In *DISC*, pages 154–169, 2015.

[39] Ricardo Jiménez-Peris, Marta Patiño-Martínez, Gustavo Alonso, and Bettina Kemme. Are quorums an alternative for data replication? *ACM Transactions on Database Systems (TODS)*, 28:257–294, 2003.

[40] Akhil Kumar. Hierarchical quorum consensus: a new algorithm for managing replicated data. *IEEE transactions on Computers*, 40, 1991.

[41] Petr Kuznetsov and Andrei Tonkikh. Asynchronous reconfiguration with byzantine failures. *arXiv preprint arXiv:2005.13499*, 2020.

[42] Leslie Lamport. On interprocess communication (part i). *Distributed Computing*, 1:77–85, 1986.

[43] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32:18–25, 2001.

[44] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, Apr 1996.

[45] Nancy A Lynch and Alexander A Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing*, pages 272–281. IEEE, 1997.

[46] Dirk Merkel et al. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014:2, 2014.

[47] Achour Mostéfaoui, Michel Raynal, and Matthieu Roy. Time-efficient read/write register in crash-prone asynchronous message-passing systems. *Computing*, 101(1):3–17, 2019.

[48] Satoshi Nakamoto and A Bitcoin. A peer-to-peer electronic cash system. *Bitcoin.–URL: https://bitcoin. org/bitcoin. pdf*, 4, 2008.

[49] Moni Naor and Avishai Wool. The Load, Capacity, and Availability of Quorum Systems. *SIAM J. Comput.*, 27:423–447, 1998.

[50] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference*, pages 305–319, 2014.

[51] Florian Oprea and Michael K Reiter. Minimizing response time for quorum-system protocols over wide-area networks. In *DSN*, pages 409–418, 2007.

[52] Behrooz Parhami. Voting algorithms: Ieee transactions on reliability. 43:617–629, 1994.

[53] David A Patterson, Garth Gibson, and Randy H Katz. A case for redundant arrays of inexpensive disks (raid). In *Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, pages 109–116, 1988.

[54] Yasushi Saito, Svend Frølund, Alistair Veitch, Arif Merchant, and Susan Spence. Fab: building distributed enterprise disk arrays from commodity components. *ACM SIGPLAN Notices*, 39:48–58, 2004.

[55] Frank B Schmuck and Roger L Haskin. Gpfs: A shared-disk file system for large computing clusters. In *FAST*, volume 2, 2002.

[56] Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[57] Stavros Souravlas and Angelo Sifaleras. Trends in data replication strategies: a survey. *International Journal of Parallel, Emergent and Distributed Systems*, 34(2):222–239, 2019.

[58] João Sousa and Alysson Bessani. *Separating the WHEAT from the Chaff: An Empirical Design for Geo-Replicated State Machines.* 2015.

[59] Alexander Spiegelman and Idit Keidar. On liveness of dynamic storage. In *International Colloquium on Structural Information and Communication Complexity*, pages 356–376, 2017.

[60] Alexander Spiegelman, Idit Keidar, and Dahlia Malkhi. Dynamic reconfiguration: A tutorial (tutorial). In *19th International Conference on Principles of Distributed Systems (OPODIS 2015)*, pages 1–14, 2016.

[61] Alexander Spiegelman, Idit Keidar, and Dahlia Malkhi. Dynamic reconfiguration: Abstraction and optimal asynchronous solution. In *DISC*, 2017.

[62] Robert H Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems (TODS)*, 4:180–209, 1979.

[63] John Turek and Dennis Shasha. The many faces of consensus in distributed systems. *Computer*, 25(6):8–17, 1992.

[64] Yuuki Ueda, Hideharu Kojima, and Tatsuhiro Tsuchiya. On the availability of replicated data managed by hierarchical voting. In *2013 International Conference on Information Science and Cloud Computing Companion*, pages 313–316. IEEE, 2013.

[65] Robbert Van Renesse and Fred B Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, volume 4, 2004.

[66] Marko Vukolić. *Quorum Systems: With Applications to Storage and Consensus.* 2012.

[67] Michael Whittaker, Aleksey Charapko, Joseph M Hellerstein, Heidi Howard, and Ion Stoica. Read-write quorum systems made practical. In *PaPoC*, pages 1–8, 2021.

[68] Faraneh Zarafshan, Abbas Karimi, Syed Abdul Rahman Al-Haddad, M Iqbal Saripan, and Shamala Subramaniam. Ancestral dynamic voting algorithm for mutual exclusion in partitioned distributed systems. *International Journal of Distributed Sensor Networks*, 9:120308, 2013.