How Hard is Asynchronous Weight Reassignment?

Hasan Heydari, Guthemberg Silvestre, Alysson Bessani

À quel point est-elle difficile la réaffectation asynchrone de poids ?

Hasan Heydari[1], Guthemberg Silvestre[2] and Alysson Bessani[1]

[1]*LASIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal*
[2]*Fédération ENAC ISAE-SUPAERO ONERA, Université de Toulouse, France*

Résumé–Les systèmes de quorum sont des abstractions fondamentales utilisées pour concevoir des protocoles distribués tolérants aux pannes et disponibles. Bien que le système de quorum majoritaire soit couramment utilisé en raison de sa simplicité et de sa tolérance optimale aux pannes, il ne parvient pas à saisir la nature dynamique et hétérogène des systèmes réels. Cette limitation devient évidente dans les services géo-distribués, où les nœuds présentent des performances hétérogènes, ou dans les services basés sur une blockchain ainsi que les systèmes de paiement décentralisés, où les enjeux ou la réputation des nœuds varient au fil du temps. La présentation d'une variante pondérée et dynamique de ce système de quorum répond efficacement à ces enjeux mais introduit un nouveau problème fondamental : à quel point est-il difficile de réattribuer les poids ? Nous répondons à cette question après avoir formalisé le problème de réaffectation de poids dans ce travail. Plus précisément, nous prouvons que la réattribution des poids est aussi difficile que la résolution d'un consensus, c'est-à-dire que cela ne peut pas être implémenté dans des systèmes distribués asynchrones sujets aux pannes.

Mots-clefs : quorum system, weighted replication, consensus, reconfiguration, blockchain

1 Introduction

Quorum systems are fundamental abstractions employed to design fault-tolerant and available distributed protocols. A quorum system is a collection of sets called *quorums* such that each quorum is a subset of processes, and every two quorums *intersect*. Although many types of quorum systems exist, such as grids [NW98] and trees [AEA90], most practical protocols (e.g., [HKJR10, OO14, L$^+$01]) utilize the regular majority quorum system (MQS) due to its simplicity and optimal fault tolerance.

In MQS, every quorum consists of a strict majority of processes. Although MQS is simple and optimally fault-tolerant, it fails to account for practical systems' dynamic and heterogeneous nature. For example, it does not consider the diverse performance of nodes in protocols executed in wide-area networks or the changing stakes or reputations of nodes in blockchain and decentralized payment systems like RepuCoin [YKDEV19]. Presenting a dynamic weighted variant of this quorum system effectively addresses these challenges but introduces a new fundamental problem : given a weighted majority quorum system [SB15, HSA21, BPRSNB22], in which each process is assigned a weight (a.k.a. vote or voting power), *how hard is it to reassign weights ?*

We answer this question after formalizing the weight reassignment problem in this work. Specifically, we prove that reassigning weights is as hard as solving consensus, *i.e.*, it cannot be implemented in asynchronous failure-prone distributed systems. We do this by demonstrating that consensus can be reduced to the weight reassignment problem, *i.e.*, a solution to the weight reassignment problem can be used to solve consensus.

2 Weight Reassignment Problem

This section presents the system model and defines the weight reassessment problem. We consider an asynchronous message-passing system composed of a finite set Π of n processes. Each process knows the set of processes. At most f processes can crash. A process is called *correct* if not crashed. Each pair of processes

is connected by a reliable communication link. No process invokes a new operation before obtaining a response from a previous one. The weighted majority quorum system (WMQS) refers to a set of quorums, each composed of processes whose total weight is greater than half of the total weight of all processes.

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

The *weight reassignment problem* aims to capture the safety and liveness properties that must be satisfied as processes' weights change over time. To formalize this problem, we first define the *change* data structure, which contains the essential information related to the outcome of a weight reassignment operation. Each process i has a local counter denoted by lc_i. We define *change* as $\Pi \times \mathbb{N} \times \Pi \times \mathbb{R}$, where the quadruple $\langle i, lc_i, j, \Delta \rangle$ indicates that the weight of process j is changed by Δ as an outcome of a reassignment request made by process i with local counter lc_i. For any change $\langle *, *, j, \Delta \rangle$, by convention, "the weight of the change" refers to Δ, and we say that "the change is created for j". We introduce a weight reassignment problem with the following operations :

— `reassign`(j, Δ), where j is a process, and Δ is a real number different from zero, and

— `read_changes`(j), where j is a process.

Each process can invoke `reassign`(j, Δ) to request changing the weight of process j by Δ. When an invocation of `reassign` is *completed* (see Definition 1), a change c corresponding to the invocation's outcome is created, and a message $\langle \text{COMPLETE}, c \rangle$ is returned to the process that invoked the operation. Any process can invoke `read_changes` to learn about the set of changes created for a process by which the weight of the process can be calculated. Each process must increment its local counter after each invocation of the `reassign` operation.

Definition 1 (*Completed* `reassign`)**.** Assume that a process i invokes `reassign`(j, Δ) when its local counter is lc_i. We say that the invocation is *completed* if there is a time after which the response of every invocation `read_changes`(j) contains a change $\langle i, lc_i, j, * \rangle$.

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

Definition 2 (Weight Reassignment Problem)**.** Any algorithm that solves the weight reassignment problem satisfies the following properties :

— Integrity. $\forall t \geq 0, \forall F \subset \Pi$ such that $|F| = f$, $\mathtt{W}_{F,t} < \frac{\mathtt{W}_{\Pi,t}}{2}$.

— Validity-I. When the `reassign`(i, Δ) operation is completed, a change $\langle *, *, i, \Delta \rangle$ is created if Integrity is not violated; otherwise, a change $\langle *, *, i, 0 \rangle$ is created.

— Validity-II. If `read_changes`(i) is invoked at time t, a set containing $\mathscr{C}_{i,t}$ is returned as the response.

— Liveness. If a correct process i invokes `reassign` (resp. `read_changes`), the invocation will eventually be completed, and i will receive a message $\langle \text{COMPLETE}, * \rangle$ (resp. a set of changes).

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

— Integrity is a consequence of Property 1, which determines the relationship between the processes' weights and f, guaranteeing the system's availability over time.

— The second property states that a change must be created as the outcome of each `reassign` invocation. It also determines how the change must be created. Note that Integrity might be violated if each invocation of `reassign`(i, Δ) is completed by creating a change $\langle *, *, i, \Delta \rangle$. Hence, to avoid the violation of Integrity, an invocation `reassign`(i, Δ) might be aborted, *i.e.*, a change $\langle *, *, i, 0 \rangle$ is created as the invocation's outcome.

— The third property determines what responses to a `read_changes` invocation are valid. Given any process that invokes `read_changes`(i) at any time $t \geq 0$, it is clear that a valid response must be as up-to-date as $\mathscr{C}_{s,t}$. Note that it is impossible to guarantee that exactly $\mathscr{C}_{s,t}$ is returned as the response due to asynchrony.

3 Impossibility Result

This section demonstrates that the weight reassignment problem cannot be implemented in asynchronous failure-prone systems. We present an insight into such an impossibility result (please refer to the extended version of this work [HSB23] for the full proof). Consider a system in which all correct processes invoke the `reassign` operation concurrently such that only one of the invocations can be completed by creating a change with non-zero weight. That is, creating two or more changes, each with non-zero weight, violates Integrity, meaning that it can make f processes have more than half of the total voting power in the system. Assume that invocations `reassign`$(1,\Delta_1),\ldots,$`reassign`(n,Δ_n) create such a situation. One can take the following steps to solve consensus among processes :

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

Since the weight of only one of the created changes is non-zero, processes can decide the same value. Consequently, consensus can be reduced to the weight reassignment problem, which means that the weight reassignment problem cannot be implemented in asynchronous failure-prone systems [Her91].

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

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

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

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

which means that Integrity is violated.

In a loop, for each process $j \in \Pi$, i repeatedly invokes `read_changes`(j) to see the invocation of which process is completed by creating a change with non-zero weight. Because of Liveness, the loop will eventually terminate. Assume that the invocation of process j is completed by creating a change $\langle j,2,j,\Delta\rangle$, where $\Delta \neq 0$. Consequently, i returns $R[j]$ as the decided value, and consensus among processes will be solved.

Theorem 1 ([HSB23])**.** Consensus can be reduced to the weight reassignment problem, *i.e.*, the weight reassignment problem cannot be implemented in asynchronous failure-prone systems.

4 Conclusion

This work studied the problem of integrating weighted majority quorums with weight reassignment protocols for any asynchronous system with a static set of processes and static fault threshold while guaranteeing

Algorithm 1 Reducing consensus to the weight reassignment problem – process i.

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

function `propose`(v_i)
1: $R[i] \leftarrow v_i$
2: **if** $i \in \{1,2,\ldots,f\}$ **then**
3: `reassign`$(i, 0.5)$
4: **else**
5: `reassign`$(i, -0.5)$
6: *decided_value* $\leftarrow \perp$
7: **repeat**
8: **for** $j \in \{1,2,\ldots n\}$ **do**
9: $\mathscr{C} \leftarrow$ `read_changes`(j)
10: **if** $\langle j,2,j,\Delta\rangle \in \mathscr{C}$ such that $\Delta \neq 0$ **then**
11: *decided_value* $\leftarrow R[j]$
12: **until** *decided_value* $\neq \perp$
13: **return** *decided_value*

availability. We showed that such a problem could not be solved in asynchronous failure-prone distributed systems. To circumvent the impossibility result presented in this work, one could refer to its extended version [HSB23], where we presented a restricted version of the weight reassignment problem called *pairwise weight reassignment*, in which weights can only be reassigned in a pairwise manner. We showed that pairwise weight reassignment could not be implemented in asynchronous failure-prone systems. We also presented a restricted version of the pairwise weight reassignment called *restricted pairwise weight reassignment* that can be implemented in asynchronous failure-prone systems. We also discussed the relation between pairwise weight reassignment and the asset transfer problem. As a case study, we presented a dynamic-weighted atomic storage based on the implementation of the restricted pairwise weight reassignment.

Références

[AEA90] Divyakant Agrawal and Amr El Abbadi. The tree quorum protocol : An efficient approach for managing replicated data. In *International Conference on Very Large Data Bases*, 1990.

[BPRSNB22] 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*, 19(3), 2022.

[Her91] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1), 1991.

[HKJR10] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. ZooKeeper : Wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, 2010.

[HSA21] Hasan Heydari, Guthemberg Silvestre, and Luciana Arantes. Efficient consensus-free weight reassignment for atomic storage. In *International Symposium on Network Computing and Applications*, 2021.

[HSB23] Hasan Heydari, Guthemberg Silvestre, and Alysson Bessani. How hard is asynchronous weight reassignment ? In *International Conference on Distributed Computing Systems*, 2023.

[L$^+$01] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32, 2001.

[NW98] Moni Naor and Avishai Wool. The load, capacity, and availability of quorum systems. *SIAM Journal on Computing*, 27(2), 1998.

[OO14] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference*, 2014.

[SB15] João Sousa and Alysson Bessani. Separating the wheat from the chaff : An empirical design for geo-replicated state machines. In *Symposium on Reliable Distributed Systems*, 2015.

[YKDEV19] Jiangshan Yu, David Kozhaya, Jeremie Decouchant, and Paulo Esteves-Verissimo. Repucoin : Your reputation is your power. *IEEE Transactions on Computers*, 68(8), 2019.