Future Generation Computer Systems 151 (2024) 260–271

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Quantised and Simulated Max–min Fairness in Blockchain Ecosystems

Serdar Metin *, Can Özturan

Boğaziçi University, Bebek, Istanbul, Turkey

ARTICLE INFO

Dataset link: https://github.com/serdarmetin/blockchainFaucet

Keywords:
Blockchain
Faucet
Max–min Fairness
Resource allocation–distribution
Quantised
Simulated

ABSTRACT

Blockchain systems heavily rely on user incentive and participation for the success and scope of their operation. For this end, *fairness* is a key precondition to attract users to join the ecosystem. In order to assure users of the fairness of the ecosystem, the distribution of shared resources must be done according to a fair and transparent policy, and it must be handled by a robust mechanism that implements this policy. The present study addresses this problem and contributes four decentralised and autonomous algorithms that may serve for the fair distribution of shared intrinsic resources. These are the adaptations of Max–min Fairness (MF), a distribution scheme which is well established in the computer science literature as fair, and its weighted version (WMF). Blockchain faucets are shared resource allocation mechanisms which accomplish this task by providing users with fixed amounts of free cryptocurrency. Faucets are employed in academic networks such as Bloxberg, and also in test networks such as Ropsten and Rinkeby We implemented both MF and WMF as blockchain faucets for the demonstration of their operation and measurement of their performance. The first algorithm, we call quantised Max–min Fairness (QMF), operates under the restriction on demand volumes but is scalable to large numbers of users, and the second, we call Simulated Max–min Fairness (SMF), operates under the restriction on the number of users but allows for various weighting policies. The other two are the weighted versions of these algorithms.

1. Introduction

Having transactions recorded on a blockchain is a shared, limited resource. For example in Ethereum networks, users access this resource in return for *gas*, a value exchanged with the blockchain's intrinsic cryptocurrency. Therefore, the distribution of this resource is coterminous with the distribution of the blockchain's intrinsic cryptocurrency. Since blockchain ecosystems rely on the user incentive to participate, *fairness* is a key factor in their operation in order to incentivise users to participate.

In commercial networks, such as the Bitcoin [1] or Ethereum Mainnet [2] cryptocurrency is either mined by costly hardware and processing time, or exchanged in return for fiat currency. Thus, accounting for the *fairness* of distribution is delegated to the external economic system: market pricing. In non-commercial blockchain ecosystems, however, the distribution should be handled by an alternative mechanism, since market pricing, by definition, is not an option.

Departing from this point, the present study focuses on the problem of fair distribution of shared resources in blockchain ecosystems. Considering the fair distribution of gas in non-commercial networks a well-suited model for studying the fair distribution problem in general, we developed solutions based on a widely employed mechanism for free cryptocurrency distribution in non-commercial networks, by adopting and adapting a fair distribution scheme into it. Namely, we adapted Max–min Fairness distribution algorithm to operate under the constraints of gas-mediated blockchain systems, within the context of a *blockchain faucet*.

Faucets are free cryptocurrency distribution mechanisms that generally offer a fixed amount of cryptocurrency in a fixed time interval. A number of examples of faucets in noncommercial blockchain ecosystems may be seen in Bloxberg infrastructure [3], or Ropsten [4] and Rinkeby [5] test networks. While this may be a simple and effective mechanism, it can be easily exploited by making recurrent demands and accumulating the obtained currency. Consequently, it cannot account for fairness of distribution.

Nevertheless, faucets are a good model of resource distribution in non-commercial networks, since it relies on user participation in the process, which we argue is the key property in blockchain programming paradigm. Thus we develop our algorithms on this model.

In the present study we built our algorithms as smart contracts, which are basically scripts stored on the blockchain and run on the given blockchains virtual machine. Although there are a number of alternatives for implementation (e.g. building the faucet as a system component into the virtual machine), our method enables easy adaptation of the faucet to varying resource types and scenarios.

* Corresponding author.
E-mail addresses: balikakli@gmail.com (S. Metin), ozturaca@boun.edu.tr (C. Özturan).

https://doi.org/10.1016/j.future.2023.08.025
Received 24 November 2022; Received in revised form 27 February 2023; Accepted 26 August 2023
Available online 6 October 2023
0167-739X/© 2023 Elsevier B.V. All rights reserved.

In other words, in addition to being a mechanism for distributing cryptocurrency, the smart contracts presented hereby can be adapted to distribute resources in general, be it intrinsic or extrinsic to the computation environment, as long as they can be represented with *integer quantities*. For example, these resources may be system resources, as presented here, or they may represent shared resources of a production line for producing different kinds of end products with common components. In the latter case it may serve for optimising the throughput, with different weights for each end product.

In a previous study [6] we addressed the problem of securing fairness, by a faucet design operating according to a distribution scheme that has been well-established in the literature for being fair, namely *Max–min Fairness* (MF). In settings where users have different demands on some shared resource, such as processing time or physical memory, MF and/or algorithms based on MF (e.g. Round Robin) has been widely used. This includes but is not limited to several operating system schedulers, network load balancers, and cloud resource managers. MF offers many desirable features, which is the reason of its wide use. Sections 2 and 3 of the present study, or the introduction of [7] may be referred to, for a broader review on MF.

We implemented MF as it is implemented in a conventional computational setting (i.e. by a central processing unit running the algorithm and carrying out the distribution process) and named it Conventional Max–min Fairness (CMF), accordingly. The tests we have run on CMF has shown that it runs into the *block gas limit exhaustion problem*, once the number of demands exceed ~ 10.

Block gas limit is a safeguard in smart contract compatible blockchain systems for preventing infinite loops. The maximum gas expenditure (consequently, the maximum number of operations) within the processing of a single block is limited to a fixed number, which is determined as a system parameter. Block gas limit exhaustion problem refers to the exceeding of an operation of this limit. Since the conventional MF implementation includes dynamic loops in its operation (described in Section 2), with growing number of demands, it eventually meets the limit and is rendered ineffective.

In the same study [6] having shown that CMF is not a suitable mechanism for blockchain ecosystems with large number of users, we offered two alternative algorithms, Autonomous Max–min Fairness (AMF) and Weighted Autonomous Max–min Fairness (WAMF), that actualises the MF scheme without relying on a central unit to carry out the iterative assignment process.

The main weakness of both AMF and WAMF is the computational burden laid on the client side. In these algorithms, the users are expected to make multiple time-critical function calls, in order to be able to claim the total shares reserved for them.

The present study contributes four novel algorithms, all of which are decentralised and autonomous, Quantised Max–min Fairness (QMF), Simulated Max–min Fairness (SMF), and their weighted versions also (i.e. WQMF and WSMF, respectively), which allow the users to claim their fair share in a single call to the claim function. As such, the algorithms we developed bear relative advantages and disadvantages to each other, each serving optimally for different use cases. These relative advantages are summarised in Table 1, and can be described as follows:

As the name implies, the QMF algorithm offers this improvement in return for a restriction on the *demand volume*. Both in AMF and WAMF, the users are allowed to make arbitrary volumes of demands, whereas in QMF the allowed demand volume is quantised. That is to say, users may only make demands in a predefined demand volume interval, and additionally in the WQMF, weights may only be assigned in a predefined weight interval. Nevertheless, as discussed in Section 5 and shown in Section 7, these intervals can be extended to reasonable sizes in order to meet the requirements of various use case scenarios.

In SMF and WSMF, on the other hand, the *number of users* that the system can efficiently support is limited, instead of demand volume. Also, there are no similar restrictions on weight in WSMF as there are in WQMF.

Table 1
Comparison of algorithms.

	Claim rounds	Volume and weights	Number of users
W/AMF [6]	Multiple	Unrestricted	Unrestricted
W/QMF (New)	Single	Quantised	Unrestricted
W/SMF (New)	Single	Unrestricted	Limited

The above mentioned restrictions allow users to calculate their respective shares, without having to carry out the costly central distribution process. As a result, W/AMF, W/QMF and W/SMF offer relative advantages and disadvantages, and each one stand optimal for different use cases.

In cases where the restrictions on demand volume and weights do not conflict with the operational requirements, W/QMF stand out to be cost-wise the most efficient as compared to their counterparts. On the other hand, if the number of users are within the limits presented here, W/SMF present the richest functionality in the most efficient way. Finally, in the cases where the burden on the client side is acceptable W/AMF works with the lightest computational cost.

The remainder of the article is organised as follows. In the following section, we overview the relevant literature. In Sections 3 and 4 we describe the MF model, and its adaptations in the present study, respectively. Following that the implementation details of contributed algorithms are laid out in Section 5. We then proceed in Section 6 to procedurally define the tests employed to measure the performance of these algorithms. In Section 7 the results of these tests are reported, in comparison with each other, and AMF and WAMF. After the results are presented, the algorithms are compared and contrasted for their relative advantages and drawbacks in Section 8. We conclude in Section 9.

2. Related work

Blockchain systems with smart contracts have been developed in many areas. These include storing and querying scientific content [8], biomedical and healthcare systems [9], decentralised voting [10], addressing the Internet [11], Internet of Things (IoT) [12], and Internet of Medical Things (IoMT), online education [13] to cite among others. For broader analysis for the structure of the smart contracts, systematic mapping studies [14] and surveys [15] may be addressed.

Blockchain systems differ from conventional systems for their operation bottleneck, and consequently the algorithms that run on these systems differ for their design and performance analysis. A number of studies have been offered for the evaluation of performance [16], principles on the algorithm design [17] and the robustness of smart contracts [18]. They all concentrate on the block gas limit exhaustion problem, which is a counterpart and a metric for the *computational cost* of a given algorithm in the conventional setting.

Max–min fairness is a widely used algorithm, that is accepted to be a fair distribution scheme in the computer engineering literature. To exemplify a few, it is used in data and communication networks [19,20], grid computation [21], for prioritising quality of service [22], and scheduling data centre jobs [23], among others.

In the recent years, resource distribution problem has been a focus of attention in the context of blockchain systems. However, these studies concentrate mainly on the distribution of resources that blockchains run on, rather than the distribution of resources on the blockchain. To name a few, [24–26] develop different models for distributing resources of fog/cloud systems among the nodes of a given blockchain. The blockchains in these studies work on IoT devices (e.g. smart vehicles), which delegate the hardware costly mining process to the fog/cloud systems at which point the resource distribution problem arises, and solved in the processing environment of the fog/cloud system manager.

The present study, in contrast, does not assume anything about the processing environment of the nodes, and carries out the distribution

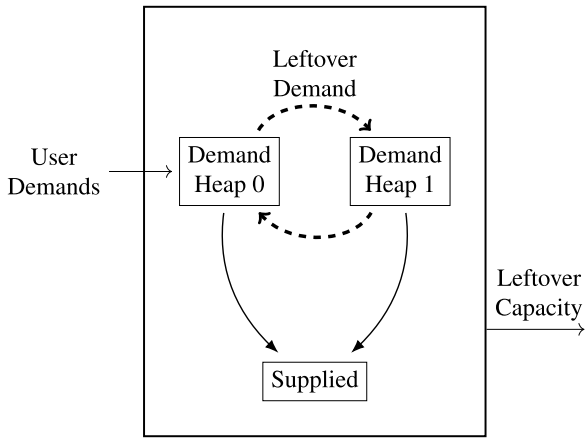

Fig. 1. Max–min Fairness Operation Diagram.

process on the virtual engine of the blockchain. To the best of our knowledge, the present study and its predecessor are the only studies in the field that address the resource distribution problem with a system working as a smart contract on the blockchain.

3. Max–min Fairness model

Before describing the models developed in the present study, we shall describe Max–min Fairness as a generic model.

Max–min Fairness (MF) is named after the principle it utilises, namely maximising the minimum share offered to any user in the demand set. Obviously, this is bounded above by the minimum demand in the set, since it is not reasonable (nor is it desirable) to assign more resources than the amount user demanded.

In order to maximise the minimum allocation, MF iterates over the demand set in an ascending order of demand volumes, and *assigns* the owner of the demand the minimum of $1/n$ of the capacity to be distributed and their demands i.e. $\min\{\frac{c}{n}, d_u\}$, where c stands for capacity, n for the number of demanders, or equivalently the cardinality of the demand set, and d_u for the demand of the user $u, u \in [1, n]$.

If there are no users with a demand volume lower than $\frac{c}{n}$ the distribution is complete in one iteration and all the demanders obtain $\frac{c}{n}$ for their fair share. If there are, however, users with a demand volume lower than $\frac{c}{n}$, at the end of the first iteration, some demands are fully satisfied and some capacity is left. MF, in turn, repeats the procedure with c' and n'. The process is illustrated in Fig. 1. User demands arrive at Heap 0. The dashed lines represent the partially satisfied demands being exchanged between the heaps. When a demand is fully supplied it is removed from this exchange loop. The arrow on the bottom right represents the leftover capacity being handed over to the next epoch.

In the weighted version of MF, at each iteration, the *unit share* is calculated by dividing the capacity to the total weight of the demanders, instead of the number of demands. The *user share*, subsequently, is calculated by multiplying the unit share with the user weight. According to this:

$$s = \frac{c}{\sum_{u=1}^{n} w_u} \tag{1}$$

and

$$s_u = s * w_u \tag{2}$$

where s stands for unit share and s_u for user share, c is capacity, n is the number of users, and w_u represents the weight of the user u.

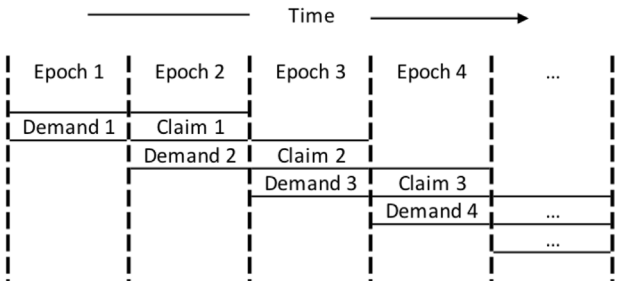

Fig. 2. Timeframe of matching demands and claims, with respect to the epochal layout.

4. Present models

In this section we will describe the working and the domain of QMF and SMF models, and also their weighted counterparts, in comparison with the conventional Max–min Fairness model. In contrast to the conventional distribution algorithm, both algorithms presented in the present study calculate and declare the maximum share that the system has to offer, so that when the users are allowed to take the minimum of their demands and this declared share (i.e. $\min\{d_u, s\}$), the capacity at hand shall not be exceeded. In turn the share is declared, the users are allowed (and expected) to assign this minimum to their balances, individually.

In the weighted versions, the same procedure is carried out for calculating a *unit share*, with which each user can obtain their individually proposed *user share* by multiplying it with their individually assigned *user weights* (i.e. $s_u = s * w_u$); and then they can take the minimum of their demand and their user share (i.e. $\min\{d_u, s_u\}$).

What differentiates QMF and SMF is the procedure each one utilises to calculate the share, which we discuss in detail in the subsequent subsections. Before moving on to describe the particular details of the two models, we will continue with their common constructs.

Both models operate in the same temporal setting. According to this, the time is fractured into *epochs*, which is defined to be a collection of a fixed number of successive blocks. For the duration of an epoch, users are allowed to make demands. At the end of an epoch, the available share is calculated according to the accumulated demands, and in the following epoch, the users can claim their fair share during the span of the epoch. The users are also allowed to make new demands to be collected in the following epoch, while claiming their demands submitted in the preceding epoch. The time overlay of demands and claims may be seen more clear in Fig. 2. The right to any share unclaimed in its due epoch is lost, and the unclaimed share is added to the capacity of the next epoch, along with the leftover capacity, if there are any.

The state of the system is accounted for by a dedicated function, `update_state`, which is called at the beginning of both the *demand* and the *claim* functions. `update_state` checks the validity of the epoch number with respect to the block number. If the epoch number needs to be updated, the capacity is replenished according to a predefined policy, and the share is recalculated. For simplicity we kept this policy in its simplest and replenished the capacity by a *constant* amount, which we refer here to as the *Epoch Capacity*. The function `update_state` does not explicitly invalidate the obsoleted demands; they are rather invalidated by the update of the *epoch* variable, by the virtue of the organisation of the remaining functions and the data structures that represent the demands.

In order to recalculate the share, `update_state` accesses a view function, `calculate_share`. The main difference between QMF and SMF, and their weighted counterparts also, is the structure of their respective `calculate_share` functions, which will be explained in detail in the relevant subsections below.

Both implementations rely on iterating over the user demands, and both get their numeric limitations over the efficiency of the loops for these iterations. The abstractions for and the layout of the demand data, in turn, determine the efficiency of these loops. In W/QMF, the loop iterates over the number of demands for the predefined demand volume interval (i.e. $[1, Quanta]$), and in W/SMF over the user demands (i.e. $\{d_1, \ldots, d_n\}$), hence the limitations on demand volume and the number of users, respectively.

4.1. Quantised Max–min Fairness model

The operation of QMF is analogous to that of *Counting Sort* algorithm, in which case to sort a collection of elements in a predefined interval, the algorithm traces the number of occurrences of each element, and enumerates the sorted list according to those counts. Likewise, QMF traces the number of demands for each demand volume, in a predefined demand volume interval, and calculates the share over these counts.

When the `calculate_share` function is called, the number of demands for each demand volume is ready, since this part is handled by the `demand` function. When a demand arrives, the number of demands for the relevant demand volume is incremented by 1, in addition to the other operations for recording the demand (e.g. updating the *demand* variable in the user list). Conveniently, demands are represented with an array, instead of a heap, since random access to demand volume counts are needed to record the increments.

The main loop of the `calculate_share` iterates over the demand array, starts by proposing 1 as the share and calculates the total capacity needed to declare the share as such. If the capacity is sufficient, the next iteration is taken, until reaching a proposal which would lead to a shortage of capacity. The loop breaks when it reaches such a proposal, and the penultimate proposal is returned to the calling function (i.e. `update_state`) to be declared as the share.

The formula for calculating the total necessary capacity for a proposal p $(1 \leq p \leq q,\ p, q \in \mathbb{Z})$ is:

$$\begin{aligned} &\sum_{i=1}^{p-1} i \cdot d_i + \sum_{j=p}^{q} p \cdot d_j \\ &= \sum_{i=1}^{p-1} i \cdot d_i + p \cdot \sum_{j=p}^{q} d_j \\ &= \sum_{i=1}^{p-1} i \cdot d_i + p \cdot (D - \sum_{i=1}^{p-1} d_i) \end{aligned} \tag{3}$$

where d_i stands for the ith entry in the demand array, and D for the total number of demands, which is collected during the demand epoch by the `demand` function. The remaining terms are calculated within the loop, each new iteration using the previous iteration's cumulative values. The first term of the equation stands for the capacity reserved for the underdemanders, and the second term for the capacity available for the overdemanders. A numerical example can be seen in Table 2.

The *Demand Volume* line represents the quantised demand volumes that are available to users, in other words they are the array indexes of the demand array, incremented by 1. In this example, the users are allowed to make demands between 1 and 7, with quanta equal to 1. The *Number of Demands (NoD)* line is an hypothetical set of demands cooked for demonstration. This is collected from users throughout an epoch in real case. The numbers indicate that 3 users demanded 1 units of the resource, 2 users demanded 2 and so on. The *Total Demand Volume (TDV)* line is the product of each demand volume with its corresponding number of demands. The following two lines are the cumulatives of the first two, respectively. The last line is calculated according to Eq. (3), and the squared value in this line represents the maximum necessary capacity that does not exceed the capacity at hand. The demand volume that corresponds to the squared value is returned to the calling function, and in turn declared as the available share.

Table 2
QMF procession example.

Demand volume	1	2	3	4	5	6	7
Number of Demands (NoD)	3	2	1	0	3	0	4
Total Demand Volume (TDV)	3	4	3	0	15	0	28
Cumulative NoD	3	5	6	6	9	9	13
Cumulative TDV	3	7	10	10	25	25	53
Necessary capacity	13	23	31	38	45	49	53

4.2. Weighted quantised Max–min Fairness model

The main difference in WQMF is that instead of a globally defined *share* for all users, we calculate *unit share* and each user's individual share is calculated by multiplying the unit share with the user's individual weight. Unit share is defined as the share reserved for unit weight (i.e. $w = 1$). According to this:

$$s_u = w_u \cdot \frac{c}{\sum_{u=1}^{n} w_u \cdot I(d_u)} \tag{4}$$

where s_u denotes the share and w_u denotes the weight of the user u, and n is the total number of users in the system. $I(x)$ is the indicator function, which returns 1 if x is a positive number, and 0 if x equals 0. In this context it allows us to indicate that only the weights of users who made a demand are included in the total sum.

In order to calculate maximum available unit share, in the `demand` function we calculate and keep the minimum unit share that suffices to satisfy the user's demand. This is given by:

$$i = \left\lceil \frac{d_u}{w_u} \right\rceil \tag{5}$$

where, i stands for the *index* to be updated in the demand array. In addition to the demand array, we also utilise a weight array, and ith entry in both arrays are incremented by their corresponding values (i.e. d_u and w_u, respectively), instead of by 1, since total demand volume and total weight values are needed in the calculation, instead of the total number of demands. Similar to QMF, the necessary capacity for declaring the unit share as p $(1 \leq p \leq q,\ p, q \in \mathbb{Z})$ is then given by:

$$\sum_{i=1}^{p-1} d_i + p \cdot \sum_{i=p}^{q} w_i \tag{6}$$

As it is in QMF, the first term gives the total supply volume satisfying the underdemanders, and the second term gives the capacity available for the overdemanders, if the unit share is to be declared as p.

In order to iteratively calculate the necessary capacity for all i and select the maximum available value, we manipulate the second term and calculate:

$$\sum_{i=1}^{p-1} d_i + p \cdot (W - \sum_{i=1}^{p-1} w_i) \tag{7}$$

where W, analogous to TD in QMF, is the total weight of the users that made a demand in the previous epoch, which we collect and calculate during the demand epoch within the `demand` function.

It should be noted that the size of the demand and weight arrays should be equal to the range of possible index values. Since the range of the available demand volumes are restricted in the $[1, q]$ interval, the algorithm needs the range of the available weight values also be a finite set to be able to operate. This brings about the further restriction that the image of the weighting function be a finite interval.

4.3. Simulated Max–min Fairness model

The operation of the `calculate_share` function of SMF is almost identical to that of the conventional Max–min Fairness algorithm. The mere difference is that the iterative assignments are replaced with a single update to a *memory* variable, which is significantly more affordable in terms of gas expenditure as compared to its *storage* counterpart, in order to calculate the maximum available share that the system has to offer to each user without exceeding the capacity. We demonstrated the difference with a slight alteration to Fig. 1, in Fig. 3. As the heaps exchange unsatisfied demands, the partial shares are *added*, as represented with the plus signs, to the *Final Share* variable. The users, then, assign the minimum of their demands and the share, individually.

The reason for the decoupling of calculating the share and assigning it to the user balances is the cost of storage write operation, as explained in Section 4. Although the share of each user is calculated during the operation, it is a better strategy in terms of gas cost, to not keep this information, and handle the individual assignments in a separate `claim` function. In fact, in a previous study [6] we implemented this model and called it Conventional Max–min Fairness (CMF), in which a central *distribute* function handles both tasks, which, as it is shown thereby, leads to rapid block gas limit exhaustion, and the system cannot support more than a few users (e.g. $n \leq 10$).

SMF iterates over the user demand vector and checks the demand of each user individually, collecting all the valid demands in a *memory heap*. This heap is a binary complete tree, implemented as an integer array on which two functions operate, one for inserting new values and the other removing the minimum element, which always reside in the tree root. These are what is called *pure* functions in the Solidity Programming Language, which do not perform neither storage write nor storage read operations, and as such, they are expected to be the least costly family of operations.

Although there are a number of alternative methods to implement a minimum heap, this stands out to be the most efficient in terms of gas cost. In fact in the same study [6], we also showed that this implementation performs slightly better in terms of gas cost, as compared to an alternative blockchain heap implementation [27], one of the few available and analysed for the average gas cost of its operations, to the best of our knowledge.

The `calculate_share` function of SMF iterates over the user demands and inserts only the demand volume to the minimum heap D_0. This is because the owner of the demand is not needed, since the assignment operation will not be handled in the `calculate_share` function. Once D_0 is populated, the remainder of the functioning is identical to MF, as indicated before, with the exception of assignment operation. The procedure is represented in Fig. 3. In the figure, *partial shares* refer to the share at each iteration, which is added to the *final share*,[1] the variable to be updated and returned to the calling function of `calculate_share`.

4.4. Weighted simulated Max–min Fairness model

In contrast with SMF, WSMF utilises a minimum heap of a node struct, rather than a simple heap of integers, to represent the demand volumes. This node struct keeps the weight of the user, in addition to the demand volumes, since the total weight is needed in the calculation of *unit share*, as explained in Section 3. In agreement with SMF, WSMF does not keep user id variable in the calculation loop. An additional difference with SMF is that, the unit share is multiplied with the user weight within the `claim` function. Other from these differences, the operation of the two algorithms are identical.

[1] In Algorithm Fig. 7, this is represented with the *result* variable.

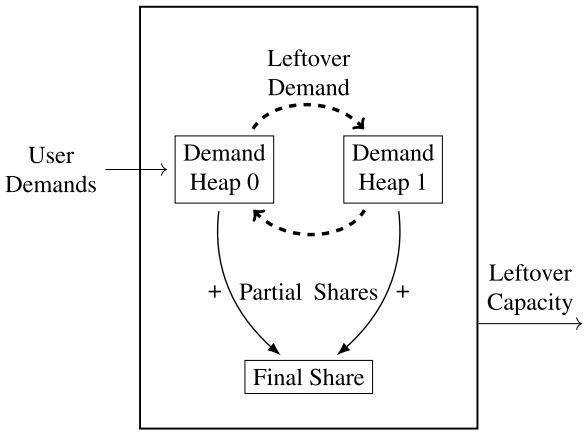

Fig. 3. Simulated Max–min Fairness Operation Diagram.

4.5. Weighting policy

In the present study we implemented and experimented W/ SMF with two different weighting policies as it will be seen in Section 7. In the first case, we chose the weights constant, randomly drawn for each user in a predefined weight interval. In the second case, we dynamically weighted each user, inversely proportional to their cumulative demand volumes, up to and including the then present demand. The first is the trivial case and it is implemented as a basis for the comparison of the added cost of calculating the dynamic weights of the second case.

The reason for our choice of this weighting policy is to incentivise the users to make the minimum demands that can satisfy their needs. It is achieved due to the fact that in this setting the most rational behaviour of the user is to keep her/his demand minimal, in order not to be disadvantageous in the long run.

For comparison, an alternative policy would be to weight the users inversely proportional to the total volume of previously allocated resources, which would lead the distribution of the total allocated volume of the resources among the users to tend to a uniform distribution in the long run. This is a matter of the needs of the system that the algorithm will be adopted to serve to.

In order to weight the users inversely proportional to the total demand volume up to and including their by then present demand, the multiplicative reciprocal of this amount is calculated. This is not all that straightforward, since Solidity offers no standard floating point variables. We addressed the same problem in [6], and here we employ the same solution developed thereby. According to this, an intermediary *precision* variable (p) is introduced to overcome the limitation, and the weights are calculated to be:

$$w_u = \left\lfloor \frac{p}{dt_u} \right\rfloor \tag{8}$$

where dt_u stands for the total demand volume of the user up to and including the then present demand. We simplify the equation and eliminate the intermediary variable. Thus, instead of directly calculating

$$s = \left\lfloor \frac{c}{\sum_{u=1}^{n} w_u} \right\rfloor \tag{9}$$

we use:

$$s = \left\lfloor \frac{c \cdot p}{\sum_{u=1}^{n} w_u} \right\rfloor \tag{10}$$

since

$$s = \left\lfloor \frac{c \cdot p}{\sum_{u=1}^{n} \frac{p}{dt_u}} \right\rfloor = \left\lfloor \frac{c}{\sum_{u=1}^{n} \frac{1}{dt_u}} \right\rfloor \tag{11}$$

In turn, while calculating the user share we use the intermediary variable p as:

$$s_u = \left\lfloor \frac{s \cdot \left\lfloor \frac{p}{dt_u} \right\rfloor}{p} \right\rfloor \tag{12}$$

The extra calculations turn out to be affordable in terms of gas consumption, as it will be seen in Section 7.

5. Implementation

In the following subsections, we will explain the implementations of QMF and SMF in detail, over the pseudocodes created for each. The reason for choosing the unweighted versions to be explained in detail is brevity. The reader might access weighted pseudocodes in Appendix A and B, which we believe will be readily intelligible once the unweighted code is examined.

We also note that the actual smart contracts, which can be accessed under the relevant folders of the Github repository at https://github.com/serdarmetin/blockchainFaucet/tree/main/Faucet [28], includes additional functions to the ones explained in the subsections below, for registering users, withdrawing currency etc., which the distribution process operates independent of. They have been implemented for convenience, and to demonstrate how the system can operate with a simple interface, and as such their performance is not a relevant metric for the overall operation of the system. Therefore, they are not included in the pseudocode, and not explained in the text.

5.1. Quantised Max–min Fairness

QMF consists of four functions, two of which the user has access to, and the other two are accessed within the former.

5.1.1. Demand

The function takes the user id and demand volume as arguments, and starts by updating the state by a call to `update_state` function. In order to select the right portion of the circular buffers, a selector variable is initiated once the state is updated. According to this, in the circular buffers, the `demand` function writes to D_0 (the vector with the index number 0) in odd epochs, and to D_1, in even epochs (line 3).

The function then proceeds to check whether the user has already made a demand in the then present epoch. If so, it returns without taking any further action, and if not, proceeds to record the demand. The variable for keeping the epoch at which the user made the last demand is updated to be the then current epoch (line 7).

In lines 8–13, the function checks whether or not the relevant entry in the demand vector has been updated in the then current epoch. If it is the first time the entry will be updated in the then present epoch, it is set to 1, and the relevant entry in the *demand reset array* (the array for keeping at which epoch the relevant entry in the demand array has been reset) is updated to be the then present epoch. If the entry is found out to be updated before, it is incremented by 1. Following that the demand volume is recorded in the user demand vector and the number of total demands is incremented by 1 (lines 14–15); then the function returns.

5.1.2. Claim

The function starts with checks and updates on the epoch variable (lines 18–25), similar to the ones in the `demand` function. In claim function, however, D_0 is used in the even epochs, and D_1 in odd ones. This alternating pattern enables `demand` and `claim` functions run in the same epochs, without interfering in each others operation. The `update_state` and the `calculate_share` functions agree with the `claim` function in the parity of their selector variables.

The function continues with updating the user claim epoch. Finally, it returns after assigning the minimum of the share and the user demand to the user account (line 27), and discounting that amount from the capacity (lines 28–29).

5.1.3. Update state

This is an *internal* function, as seen above, called by `de- mand` and `claim` functions. It is mainly responsible for updating the epoch (lines 32–33), and if the epoch needs to be updated, *capacity* (line 34), *share* (line 35), and *totalDemands* (line 36) variables along with it. The due epoch number is calculated by subtracting the number of the block that the contract was deployed (*offset*) from the then current block number, dividing it by the epoch span, and finally taking the floor of the resulting number (line 36).

5.1.4. Calculate share

This function is accessed only within the `update_state` function, thus it assumes the state to be up-to-date, and immediately starts with initiating the selector variable, which, as mentioned above, agrees with the selector variable of the claim function.

Having initiated the selector variable, the function initiates two more local variables. These variables are used to keep the cumulative number of demands and the cumulative demand volume, as the share is calculated iteratively, thus the names of the variables: *cumulativeDemands* and *cumulativeDemandVolume*.

Lines 45–59 show the main loop of the `calculate_share` function, which at each iteration, calculates the cost of declaring the share as equal to the number of the iteration. That is to say, in first iteration the cost of declaring the share as 1 is calculated, in second iteration 2, and so on, up to the maximum allowed demand volume, *Quanta*. If at any step the cost exceeds the available capacity, the loop breaks, returning the penultimate proposal as the share. If the loop finishes without breaking, the value *Quanta* is returned.

Since the function keeps the cumulative values in the local variables, the write function is not costly. The main cost is due to the storage reads in lines 49 and 52, which is still affordable, as shown in Section 7 (see Figs. 4–7).

5.2. Simulated Max–min Fairness

Like QMF, SMF also consists of four functions, two of which the user has access to, and the remaining two is accessed within the former.

5.2.1. Demand

The function takes the user id and demand volume as arguments, and starts with updating the state by a call to `update_ state` function. In order to select the right portion of the circular buffers, a selector variable is initiated after the state is updated. The `demand` function writes to D_0 in odd epochs, and to D_1, in even epochs (line 3).

Next is to check whether the user has already made a demand in the then present epoch. If so, the function returns, and if not, moves on to record the demand (line 7). Lastly, the variable for keeping the epoch at which the user made the last demand is updated to be the then current epoch (line 8), and the function returns.

5.2.2. Claim

The function starts with a call to the `update_state` function, and initiates the selector variable. In this claim function also, D_0 is used in the even epochs, and D_1 in odd ones, like it is in the claim function of QMF.

The function continues with updating the user claim epoch. Finally, it returns after assigning the minimum of the share and the user demand to the user account (lines 20–21), and discounting that amount from the capacity (lines 22–23).

5.2.3. Update state

This is an *internal* function, as seen above, called by `demand` and `claim` functions. It is mainly responsible for updating the epoch (line 27); and if the epoch needs to be updated, *capacity* (line 28), and *share* (line 29) variables along with it.

```
1:  procedure DEMAND(User, Volume)      ▷ Make a Demand
2:      UPDATE_STATE();
3:      selector ← (Epoch + 1) (mod 2);
4:      if User.demandEpoch[selector] = Epoch then
5:          return;
6:      end if
7:      User.demandEpoch[selector] ← Epoch;
8:      if ResetEpoch[selector][Volume] ≠ Epoch then
9:          ResetEpoch[selector][Volume] ← Epoch;
10:         Demands[selector][Volume] ← 1;
11:     else
12:         Demands[selector][Volume] + +;
13:     end if
14:     User.demand[selector] ← Volume;
15:     TotalDemands + +;
16: end procedure

17: procedure CLAIM(User)               ▷ Claim User Share
18:     UPDATE_STATE();
19:     selector ← Epoch (mod 2);
20:     if User.demandEpoch[selector] ≠ Epoch − 1 then
21:         return;
22:     end if
23:     if User.claimEpoch = Epoch then
24:         return;
25:     end if
26:     User.claimEpoch ← Epoch;
27:     User.balance ← min(Share, User.demand[selector]);
28:     Capacity ← Capacity−
29:                    min(Share, User.demand[selector]);
30: end procedure

31: procedure UPDATE_STATE()
32:     if Epoch ≠ (BlockNumber−Offset)/EpochSpan then
33:         Epoch ← (BlockNumber−Offset)/EpochSpan;
34:         Capacity ← Capacity + EpochCapacity;
35:         Share ← CALCULATE_SHARE();
36:         TotalDemands[(Epoch + 1) (mod 2)] ← 0;
37:     end if
38: end procedure
```

Fig. 4. QMF Pseudocode.

```
39: procedure CALCULATE_SHARE()
40:     selector ← Epoch (mod 2);
41:     cumulativeDemands ← 0;
42:     cumulativeDemandVolume ← 0;
43:     for i ← 1, Quanta do
44:         if ResetEpoch[selector][i] = Epoch − 1 then
45:             cumulativeDemands ←
46:                                 cumulativeDemands+
47:                                 Demands[selector][i];
48:             cumulativeDemandVolume ←
49:                             cumulativeDemandVolume+
50:                                 i ∗ Demands[selector][i];
51:         end if
52:         if Capacity < cumulativeDemandVolume+
53:                             i ∗ (TotalDemands[selector]−
54:                                 cumulativeDemands) then
55:             return i − 1;
56:         end if
57:     end for
58:     return Quanta;
59: end procedure
```

Fig. 5. QMF Pseudocode Continued.

5.2.4. Calculate share

This function uses 5 local variables, thus starts with initiating them. First is the selector variable. Next is the heap, which is used to simulate the demand heaps in the conventional algorithm. In order to keep the global *capacity* variable unaltered, a local variable with the name *simulatedCapacity* is used instead. In order to keep the temporary share in between the iterations, another local variable *simulatedShare* is used, and accumulating shares are collected in the local variable *result*, in order to be returned in the final.

There are two main loops in the algorithm. The first loop (lines 38–43) is responsible for reading the user demands from the demand vector (storage variable), and if the demand is valid (i.e. recorded in the immediately previous epoch, lines 39–40) writing to the local heap. This means two storage reads (one for *demand epoch* and one for *demand volume*) and a single memory write, the former of which is relatively costly.

Having prepared the local heap, in line 46, the simulated share of the first iteration is calculated. Lines 45–59 show the second main loop of the function. The loop runs until either the heap has been emptied (which means that all the demands are satisfiable with the capacity at hand) or the capacity is less than the number of demands.

Two additional heaps are nested within this loop. In lines 47–51 the demands that are fully satisfiable, in other words, the demands that are less than or equal to the simulated capacity of the then present iteration, are deducted from the capacity, and the demand, being fully satisfied, is removed from the heap. The loop breaks if and when it encounters the first demand that is greater than the simulated share, since they can only be offered as much as the simulated share.

Instead of taking each demand and deducing one simulated share from the capacity for each, the remaining number of demands is multiplied with the simulated share, and that total is deducted from the capacity in a single step (lines 52–53). The second nested loop (lines 54–56), in turn, iterates over the local demand heap, and deducts simulated share from the remaining demands. The simulated share is cumulated in the result variable (line 57), and then recalculated for the next loop (line 58). When the outer loop terminates, the result variable is returned (line 62) to the `update_state` function.

The functions to insert to and remove from the local heap are what is called *pure* functions in Solidity. They do not read from storage variables, in addition to not writing on them, which renders this category of functions the least costly. The main cost stems from the first outer loop, leading to the limitation on the number of demands, and consequently, on the number of users.

6. Procedure and parameters

We tested our algorithms in a local blockchain, operated by Parity Ethereum 2.7.2 [29], and we implemented the smart contracts in Solidity 0.5.13. The block gas limit we assume is $8,000,000$.[2]

Parity implementation of Ethereum offers customisable consensus protocols. Among those is the so called *instant seal engine*, which places each transaction into an individual block of its own. The engine is

[2] The real value is dynamically set in each system by the collective contribution of its miners.

```
1: procedure DEMAND(User, Volume)        ▷ Make a Demand
2:     UPDATE_STATE();
3:     selector ← (epoch + 1) (mod 2);
4:     if User.demandEpoch[selector] = Epoch then
5:         return;
6:     end if
7:     User.demand[selector] ← Volume;
8:     User.demandEpoch[selector] ← Epoch;
9: end procedure

10: procedure CLAIM(User)                ▷ Claim User Share
11:     UPDATE_STATE();
12:     selector ← epoch (mod 2);
13:     if User.demandEpoch[selector] ≠ Epoch − 1 then
14:         return;
15:     end if
16:     if User.claimEpoch = Epoch then
17:         return;
18:     end if
19:     User.claimEpoch ← Epoch;
20:     User.balance ←
21:                 min(Share, User.demand[selector]);
22:     Capacity ←
23:         Capacity − min(Share, User.demand[selector]);
24: end procedure

25: procedure UPDATE_STATE()
26:     if Epoch ≠ (BlockNumber−Offset)/EpochSpan then
27:         Epoch ← (BlockNumber−Offset)/EpochSpan;
28:         Capacity ← Capacity + EpochCapacity;
29:         Share ← CALCULATE_SHARE();
30:     end if
31: end procedure
```

Fig. 6. SMF Pseudocode.

```
32: procedure CALCULATE_SHARE()
33:     selector ← Epoch (mod 2);
34:     heap ← ∅;
35:     simulatedCapacity = Capacity;
36:     simulatedShare ← 0;
37:     result ← 0;
38:     for i ← 1, NumberOfUsers do
39:         if User.DemandEpoch[selector] =
40:                                         Epoch − 1 then
41:             INSERT(heap, User.demand[selector]);
42:         end if
43:     end for
44:     simulatedShare ← ⌊simulatedCapacity / heapSize⌋;
45:     while heap.length > 0 &
46:                 simulatedCapacity ≥ heap.length do
47:         while heap[0] < simulatedShare do
48:             simulatedCapacity ←
49:                         simulatedCapacity − heap[0];
50:             DELETEMIN(heap);
51:         end while
52:         simulatedCapacity ← simulatedCapacity−
53:                         simulatedShare ∗ heap.length;
54:         for i = 0, heap.length do
55:             heap[i] ← heap[i] − simulatedShare;
56:         end for
57:         result ← result + simulatedShare;
58:         simulatedShare ← ⌊simulatedCapacity / heap.length⌋;
59:     end while
60:     return result;
61: end procedure
```

Fig. 7. SMF Pseudocode Continued.

specifically designed for contract development, since the block preparation and addition latency (*block latency*, colloquially) is rarely a relevant parameter in the development and verification processes of the algorithms, at least for the time being.

For our case, the instant seal engine also allows us operationalise *time* in terms of number of blocks, and define the *epoch span* in terms of it. As such, our results are generalisable to every blockchain environment, independent of the consensus algorithms and parameters they employ.

In the tests we run to measure the performance of our implementations, we parameterised the test parameters over a single one among them. That is the performance bottleneck variable, which in the cases of QMF and WQMF is the *Quanta* value, and in SMF and WSMF is the *number of users*. We have run our tests with 10, 50, 100, 250, 500, and 1000 quanta values for W/QMF, and with 10, 50, 100, 250, 500, and 1000 users for W/SMF, and observed in each case how the cost scales with these growing values. The chosen values for each parameter can be seen more explicitly in Tables 3 and 4 for W/QMF and W/SMF, respectively.

The tests are run for 3 sets, extended over 4 epochs. According to this, in the first epoch, users are registered for n blocks, and each one made a demand, taking n more blocks, concluding the first epoch. The remaining 3 epochs followed the pattern of first claiming the pending demands from the previous epoch for n blocks, and making new demands for the next epoch for n more blocks. Averages of each set are collected, and averaged out for the final result to be reported.

Table 3
The parameters and their values used in the tests for QMF and WQMF.

Parameter	Value	Definition
Quanta	Q	Maximum demand volume
Number of users	1,000	The number of users in the system
Epoch capacity	$500 \cdot Q$	The amount to be distributed for each epoch
Epoch span	2000	The duration of an epoch in number of blocks
Demand interval	$[1, Q]$	The interval which the demands are uniformly drawn from
Weight interval	$[1, 10]$	The interval which the weights are uniformly drawn from

In W/QMF, the demands are drawn from a discrete uniform distribution in $[1, Q]$ interval, and the capacity is set to $500 \cdot Q$, which is slightly less than the expected average ($\mathbb{E}[d_u] = \frac{1+Q}{2}, u \in U$) of the uniform distribution in the interval $[1, Q]$. We deliberately introduced this shortage in order for the tests to allow the cases where the total volume of the demands exceed the capacity at hand. No further shortage or abundance of resources is forced into the tests by other parameters. Similarly, in W/SMF, the demands are drawn from the $[15, 35)$ discrete interval uniformly, and the capacity is set to $20 \cdot n$, offering each user slightly less than the expected average ($\mathbb{E}[d_u] = 24.5, u \in U$), in order to allow the tests to include capacity exceeding total demand volume cases.

Table 4
The parameters and their values used in the tests for SMF and WSMF.

Parameter	Value	Definition
Number of users	n	The number of users in the system
Epoch capacity	$20n$	The amount to be distributed for each epoch
Epoch span	$2n$	The duration of an epoch in number of blocks
Demand interval	$[15, 35)$	The interval which the demands are uniformly drawn from
Weight interval	$[1, 10]$	The interval which the weights are uniformly drawn from

Table 5
Average gas cost for demand.

q	QMF	WQMF	n	SMF	WSMF-C	WSMF-R	AMF	WAMF
10	66,751	74,544	10	65,101	60,161	75,837	70,245	79,732
50	67,754	75,878	50	65,101	60,161	75,837	67,351	77,135
100	69,008	77,150	100	65,101	60,161	75,837	66,989	76,835
250	72,701	79,902	250	65,101	60,161	75,837	–	–
500	77,743	83,066	500	65,101	–	–	66,700	71,365
1,000	84,817	88,440	1,000	65,101	–	–	–	–

Table 6
Average gas cost for claim.

q	QMF	WQMF	n	SMF	WSMF-C	WSMF-R	AMF	WAMF
10	56,122	56,721	10	56,142	56,641	57,031	46,800	46,643
50	56,121	56,719	50	56,142	56,639	57,533	42,240	44,852
100	56,120	56,720	100	56,142	56,640	57,532	42,114	44,763
250	56,120	56,719	250	56,142	56,641	57,531	–	–
500	56,119	56,719	500	56,142	–	–	42,047	45,319
1,000	56,119	56,719	1,000	56,142	–	–	–	–

In the cases of constant weights, the weights are drawn from a uniform distribution in the $[1, 10]$ discrete interval.

7. Results

We summarise the results in three tables, in which we present average cost for the `demand` (Table 5), average cost for the `claim` (Table 6), and maximum cost for the `update_state` (Table 7) functions. The reason for preferring maximum values instead of average in the latter case is that it is more convenient to consider the worst case scenarios rather than the average case, since this is the main bottleneck in all the algorithms. Moreover, the distribution of the cost of this function is widely skewed, the first invocation at each epoch being several orders of magnitude larger than the remaining invocations of the function, rendering the arithmetic average misrepresentative. Finally, for reasons of fairness, this cost is refunded to the user, since being the first to invoke a `claim` or a `demand` in a given epoch is hardly a burden that may be fairly loaded on a single random user. Thus, it is not really a cost for the users to shoulder, but rather for the system itself, and the only important concern for this cost is to keep it within the boundaries of block gas limit.

As seen in Tables 5 and 6, the cost of demand and claim functions are contained well within the block gas limit, being 2 orders of magnitude below it, and showing low variability. In fact, the cost of the `demand` function for W/SMF is constant (i.e. $\sigma = 0$) for both within and between the trials, and between the tests with different numbers of users.

As for the `update_state` maximums, Table 7 reveals, and Fig. 8 illustrates, the growth of the cost tends to linear, and the reported values are well contained within the block gas limit. The missing values in Table 7 are due to the fact that, WSMF exceeds the block gas limit for these number of user values, and the presented tests in the previous study [6] were considered sufficient in W/AMF.

Table 7
Maximum gas cost for update state.

q	QMF	WQMF	n	SMF	WSMF-C	WSMF-R
10	61,083	56,528	10	79,859	141,159	114,289
50	100,353	109,094	50	237,336	693,835	724,976
100	163,878	128,952	100	450,576	1,480,234	1,438,054
250	317,493	404,156	250	1,277,618	5,247,693	4,831,477
500	602,169	714,624	500	2,611,722	–	–
1,000	1,087,829	1,285,120	1,000	5,257,236	–	–

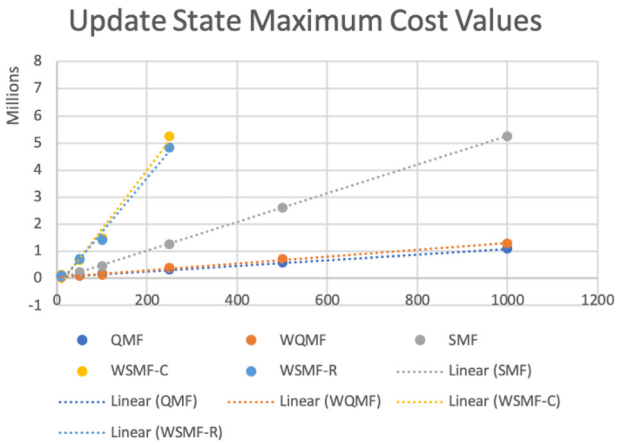

Fig. 8. Maximum Cost Values for the Update State Function with Linear Regression.

8. Discussion

As the results suggest, the `demand` and `claim` functions show low variability among the algorithms, and all are efficient with respect to the block gas limit, being several orders of magnitude below it.

It should be noted that the total cost of the `claim` function in W/AMF is obtained by multiplying the values presented in Table 6 by the number of calls to the function with the necessary number of calls, which is a function of the distribution of the demands, and which may vary among the users with different demands, since in W/AMF total claim process may take more than a single call in each epoch.

The maximum number of users we reached that WSMF can support under $8,000,000$ block gas limit is 250. Nevertheless, it should be noted that the algorithm can be optimised further in the low level in order to decrease the cost (e.g. reduce the size of the variables) and allow for higher numbers of users. We did not undertake such an endeavour for two reasons:

First, the main aim and the scope of the present study is to demonstrate the cost *structure*, rather than to provide tight bounds for the cost.

Second, the exact cost of the operation of the `calculate _share` (and consequently the `update_state`) is a function of the distribution of the demands. That is because the number of iterations needed for the distribution process to complete is dependent on how the demands are distributed. For example, in the extreme cases where all demands are below or all demands are above the available average (i.e. $\leq \frac{c}{n}$, or $\geq \frac{c}{n}$) the algorithm takes a single iteration, assigning each user their demands in the former case, and $\frac{c}{n}$ in the latter.

As it is also mentioned in the introduction, in cases where the restrictions on demand volume and weights do not conflict with the

```
1:  procedure DEMAND(User, Volume)        ▷ Make a Demand
2:      UPDATE_STATE();
3:      selector ← (epoch + 1) (mod 2);
4:      if User.demandEpoch[selector] = Epoch then
5:          return;
6:      end if
7:      User.demandEpoch[selector] ← Epoch;
8:      index ← ⌈Volume / User.weight⌉;
9:      if ResetEpoch[selector][index] ≠ Epoch then
10:         ResetEpoch[selector][index] ← Epoch;
11:         Demands[selector][index] ← Volume;
12:         Weights[selector][index] ← User.Weight;
13:     else
14:         Demands[selector][index] ←
15:                     Demands[selector][index] + Volume;
16:         Weights[selector][index] ←
17:                   Weights[selector][index] + User.weight;
18:     end if
19:     User.demand[selector] ← Volume;
20:     TotalDemands ← TotalDemands + Volume;
21:     TotalWeights ← TotalWeights + User.weight;
22: end procedure

23: procedure CLAIM(User)                 ▷ Claim User Share
24:     UPDATE_STATE();
25:     selector ← Epoch (mod 2);
26:     if User.demandEpoch[selector] = Epoch then
27:         return;
28:     end if
29:     if User.claimEpoch = Epoch then
30:         return;
31:     end if
32:     User.claimEpoch ← Epoch;
33:     share ← User.weight ∗ UnitShare;
34:     User.balance ← min(share, User.demand[selector]);
35:     Capacity ←
36:           Capacity − min(share, User.demand[selector]);
37: end procedure
```

Fig. A.9. WQMF Pseudocode.

```
38: procedure UPDATE_STATE()
39:     if Epoch ≠ (BlockNumber−Offset) / EpochSpan then
40:         Epoch ← (BlockNumber−Offset) / EpochSpan;
41:         Capacity ← Capacity + EpochCapacity;
42:         UnitShare ← CALCULATE_UNIT_SHARE();
43:         TotalDemands[(Epoch + 1) (mod 2)] ← 0;
44:     end if
45: end procedure

46: procedure CALCULATE_UNIT_SHARE()
47:     selector ← Epoch (mod 2);
48:     cumulativeDemands ← 0;
49:     cumulativeWeights ← 0;
50:     for i ← 1, Quanta do
51:         if ResetEpoch[selector][i] = Epoch − 1 then
52:             cumulativeDemands ←
53:                                   cumulativeDemands+
54:                                   Demands[selector][i];
55:             cumulativeWeights ←
56:                                    cumulativeWeights+
57:                                    Weights[selector][i];
58:         end if
59:         if Capacity < cumulativeDemands + i∗
60:                                              (totalWeights−
61:                                  cumulativeWeights) then
62:             return i − 1;
63:         end if
64:     end for
65:     return Quanta;
66: end procedure
```

Fig. A.10. WQMF Pseudocode Continued.

operational requirements, W/QMF stand out to be cost-wise most efficient as compared to their counterparts. As we have shown, the interval in which users are allowed to make demands might be stretched up to 1000 quanta with relative ease, since the gas cost in such a case is $1,087,829$ for QMF and $1,285,120$ for WQMF (Table 7).

On the other hand, if the number of users are within the limits presented here, W/SMF present the richest functionality in the most efficient way. In contrast to W/QMF, dynamic setting of the weights is possible, which in our example enabled us to implement a weighting policy that weights each user negatively proportional to her cumulative past demands. This policy can account for the *long term fairness* of the distribution, which have not been possible in the WQMF setting. The setting has been able to be run with up to 250 users (Table 7), and for higher number of users, extra capacity may be freed in return for smaller variable sizes. In most of the cases the default 256 bit integers shall far more than enough, but this also is dependent on the projected life cycle of the system to be implemented.

Finally, in the cases where the burden on the client side is acceptable W/AMF works with the lightest computational cost. In addition, the dynamic weighting policy is also available for this algorithm.

9. Conclusion

The present study focused on the problem of fairly distributing shared resources on blockchains. We developed 4 solutions to the problem at hand, with relative advantages and drawbacks. This is a follow up of a previous study where we had developed 2 algorithms to address the same question. As a result, we ended up offering 6 algorithms in total, 2 in the previous, and 4 in the present study, each of which is an optimal solution for different scenarios. To the best of our knowledge, these are the only studies in the field that address the resource distribution problem in the context of the procession of blockchain systems.

The common improvement of the presently developed algorithms on their predecessor is that they ease the burden on the client side of what we may call a *client-blockchain architecture*, by reducing the necessary number of transactions for each user to complete a distribution period. While the previously developed algorithms need multiple transactions to fully allocate each user's reserved share, the present ones complete the allocation in a single transaction.

The improvement these algorithms offer, however, comes with different trade-offs for each one. While 2 of these algorithms run under the restriction of a predefined demand interval, the other 2 can support a

```
1:  procedure DEMAND(User, Volume)        ▷ Make a Demand
2:      UPDATE_STATE();
3:      selector ← (epoch + 1) (mod 2);
4:      if User.demandEpoch[selector] = Epoch then
5:          return;
6:      end if
7:      User.demand[selector] ← Volume;
8:      User.demandEpoch[selector] ← Epoch;
9:      User.totalDemand ← User.totalDemand+
10:                                        Volume;
11: end procedure

12: procedure CLAIM(User)                 ▷ Claim User Share
13:     UPDATE_STATE();
14:     selector ← Epoch (mod 2);
15:     if User.demandEpoch[selector] = Epoch then
16:         return;
17:     end if
18:     if User.claimEpoch = Epoch then
19:         return;
20:     end if
21:     User.claimEpoch ← Epoch;
22:     if User.demandEpoch[1 − selector] = Epoch then
23:         Share ← UnitShare∗
24:                 ⌊Precision / (User.totalDemand−User.demand[selector])⌋;
25:     else
26:         share ← UnitShare ∗ ⌊Precision / User.totalDemand⌋;
27:     end if
28:     User.balance ←
29:                 min(share, User.demand[selector]);
30:     Capacity ← Capacity−
31:                 min(share, User.demand[selector]);
32: end procedure
```

Fig. B.11. WSMF Pseudocode.

```
33: procedure CALCULATE_UNIT_SHARE()
34:     selector ← Epoch (mod 2);
35:     heap[0] ← ∅;                      ▷ Initiate Empty Heaps
36:     heap[1] ← ∅;
37:     simulatedCapacity = Capacity ∗ Precision;
38:     simulatedShare ← 0;
39:     simulatedUnitShare ← 0;
40:     totalWeight ← 0;
41:     result ← 0;
42:     for i ← 1, NumberOfUsers do
43:         if User[i].demandEpoch[selector] =
44:                                     Epoch − 1 then
45:             userWeight ← ⌊Precision / User.totalDemand⌋
46:             node ←
47:                     {User.demand[selector], userWeight}
48:             INSERT(heap[0], node);
49:             totalWeight ← totalWeight + userWeight;
50:         end if
51:     end for
52:     while heap[selector].length > 0
53:                         & simulatedCapacity ≥
54:                                 totalWeight do
55:         simulatedUnitShare ← ⌊simulatedCapacity / totalWeight⌋;
56:         result ← result + simulatedShare;
57:         while heap[selector].length > 0 do
58:             simulatedShare ←
59:                         heap[selector][0].weight∗
60:                             simulatedUnitShare;
61:             if simulatedShare = 0 then
62:                 totalWeight ← totalWeight−
63:                             heap[selector][0].weight;
64:                 DELETEMIN(heap[selector])
65:             else if heap[selector][0].volume ≤
66:                             simulatedShare then
67:                 simulatedCapacity ←
68:                             simulatedCapacity−
69:                         heap[selector][0].volume;
70:                 DELETEMIN(heap[selector]);
71:             else
72:                 node ← {heap[selector][0].volume−
73:                             simulatedShare,
74:                         heap[selector][0].weight};
75:                 DELETEMIN(heap[selector]);
76:                 INSERT(heap[1 − selector], node);
77:             end if
78:             selector ← 1 − selector;
79:         end while
80:     end while
81:     return result;
82: end procedure
```

Fig. B.12. WSMF Pseudocode Continued.

limited number of users. Nevertheless, even under these restrictions, these algorithms present rich use cases for various scenarios, which are numerically measured, analysed, and presented in the relevant sections above.

Although as a general programming principle it is preferable to not have loops in algorithms which are intended to run on blockchains, well-constrained loops offer their advantages. The present study may serve as exemplary to the cost growth structure of loops, and to identify which elements to consider for saving on the gas cost, in order to develop more efficient algorithms. This may include the selection of assembly level operations, variable sizes and memory locations, among others, for each are heterogeneously charged for gas. Prospective studies may focus on those points to provide better-detailed numeric analyses on the issue, and establish better-defined and better-quantified programming principles for blockchain systems.

The algorithms presented hereby may also serve as foundations for developing a system for the fair distribution of multiple and heterogeneous resources. For example Dominant Resource Fairness (DRF) [7], an algorithm developed to handle such cases, uses Max–min Fairness as its subroutine, and the algorithms developed in the present study can be used to adapt DRF to the blockchain systems. To what extent and to which efficiency is a question we leave for the future work.

CRediT authorship contribution statement

Serdar Metin: Conceptualisation, Methodology, Software, Validation, Formal analysis, Investigation, Resources, Data curation, Writing – original draft, Visualisation. **Can Özturan:** Review & editing, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The code and the data can be found at the depository: https://github.com/serdarmetin/blockchainFaucet.

Acknowledgements

The authors thank Professor Yağmur Denizhan of Boğaziçi University Electrics and Electronics Department for her contributions of insight, and Professor Cem Ersoy of Boğaziçi University Computer Engineering Department for his contributions in procuring a dynamic and liberated research environment, that is Boğaziçi University Computer Engineering Department.

Appendix A. Pseudocode of WQMF

See Figs. A.9 and A.10.

Appendix B. Pseudocode of WSMF

See Figs. B.11 and B.12.

References

[1] S. Nakamoto, et al., Bitcoin: A peer-to-peer electronic cash system, 2008.
[2] G. Wood, et al., Ethereum: A secure decentralised generalised transaction ledger, Ethereum Project Yellow Pap. 151 (2014) (2014) 1–32.
[3] F. Kleinfercher, S. Vengadasalam, J. Lawton, Bloxberg whitepaper, the trusted research infrastructure, v1.1, 2020, https://bloxberg.org/wp-content/uploads/2020/02/bloxberg_whitepaper_1.1.pdf. (Online; Accessed 1 March 2020).
[4] Ropsten Ethereum (RETH) faucet. URL https://faucet.dimensions.network/.
[5] Rinkeby authenticated faucet. URL https://faucet.rinkeby.io/.
[6] S. Metin, C. Özturan, Max–min fairness based faucet design for blockchains, Future Gener. Comput. Syst. 131 (2022) 18–27.
[7] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, I. Stoica, Dominant resource fairness: Fair allocation of multiple resource types, in: Nsdi, vol. 11, (no. 2011) 2011, p. 24.
[8] G. Gürsoy, C.M. Brannon, M. Gerstein, Using ethereum blockchain to store and query pharmacogenomics data via smart contracts, BMC Med. Genomics 13 (1) (2020) 1–11.
[9] T.-T. Kuo, H.-E. Kim, L. Ohno-Machado, Blockchain distributed ledger technologies for biomedical and health care applications, J. Am. Med. Inform. Assoc. 24 (6) (2017) 1211–1220.
[10] N. Kshetri, J. Voas, Blockchain-enabled e-voting, Ieee Softw. 35 (4) (2018) 95–99.
[11] S. Angieri, A. García-Martínez, B. Liu, Z. Yan, C. Wang, M. Bagnulo, A distributed autonomous organization for internet address management, IEEE Trans. Eng. Manage. 67 (4) (2020) 1459–1475, http://dx.doi.org/10.1109/TEM.2019.2924737.
[12] W. Wang, C. Qiu, Z. Yin, G. Srivastava, T.R. Gadekallu, F. Alsolami, C. Su, Blockchain and PUF-based lightweight authentication protocol for wireless medical sensor networks, IEEE Internet Things J. (2021).
[13] A. Alam, Platform utilising blockchain technology for elearning and online education for open sharing of academic proficiency and progress records, in: Smart Data Intelligence: Proceedings of ICSMDI 2022, Springer, 2022, pp. 307–320.
[14] D. Macrinici, C. Cartofeanu, S. Gao, Smart contract applications within blockchain technology: A systematic mapping study, Telemat. Inform. 35 (8) (2018) 2337–2354, http://dx.doi.org/10.1016/j.tele.2018.10.004.
[15] S. Rouhani, R. Deters, Security, performance, and applications of smart contracts: A systematic survey, IEEE Access 7 (2019) 50759–50779.
[16] M. Alharby, A. van Moorsel, Blocksim: A simulation framework for blockchain systems, ACM SIGMETRICS Perform. Eval. Rev. 46 (3) (2019) 135–138.
[17] L. Marchesi, M. Marchesi, G. Destefanis, G. Barabino, D. Tigano, Design patterns for gas optimization in ethereum, in: 2020 IEEE International Workshop on Blockchain Oriented Software Engineering, IWBOSE, IEEE, 2020, pp. 9–15.
[18] G. Canfora, A. Di Sorbo, S. Laudanna, A. Vacca, C.A. Visaggio, Gasmet: Profiling gas leaks in the deployment of solidity smart contracts, 2020, arXiv e-prints, arXiv–2008.
[19] E.L. Hahne, Round-robin scheduling for max-min fairness in data networks, IEEE J. Selected Areas Commun. 9 (7) (1991) 1024–1039.
[20] D. Nace, M. Pióro, Max-min fairness and its applications to routing and load-balancing in communication networks: A tutorial, IEEE Commun. Surv. Tutor. 10 (4) (2008) 5–17.
[21] R. Gogulan, A. Kavitha, U.K. Kumar, Max min fair scheduling algorithm using in grid scheduling with load balancing, Int. J. Res. Comput. Sci. 2 (3) (2012) 41.
[22] P. Marbach, Priority service and max-min fairness, in: Proceedings. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies, Vol. 1, IEEE, 2002, pp. 266–275.
[23] A. Ghodsi, M. Zaharia, S. Shenker, I. Stoica, Choosy: Max-min fair sharing for datacenter jobs with constraints, in: Proceedings of the 8th ACM European Conference on Computer Systems, 2013, pp. 365–378.
[24] J. Zhang, W. Lou, H. Sun, Q. Su, W. Li, Truthful auction mechanisms for resource allocation in the internet of vehicles with public blockchain networks, Future Gen. Comput. Syst. 132 (2022) 11–24.
[25] H. Qiu, T. Li, Auction method to prevent bid-rigging strategies in mobile blockchain edge computing resource allocation, Future Gener. Comput. Syst. 128 (2022) 1–15.
[26] Y. Jiao, P. Wang, D. Niyato, K. Suankaewmanee, Auction mechanisms in cloud/fog computing resource allocation for public blockchain networks, IEEE Trans. Parallel Distrib. Syst. 30 (9) (2019) 1975–1989.
[27] Z. Mitton, Priority Queue on Ethereum: eth-heap, 2018, URL https://github.com/zmitton/eth-heap.
[28] S. Metin, blockchainFaucet, 2020, URL https://github.com/serdarmetin/blockchainFaucet.
[29] Paritytech, parity ethereum, 2019, https://github.com/paritytech/parity-ethereum. (Online; Accessed 19 November 2019).

Serdar Metin is a graduate of Psychology (2007), specialised in Neuroscience. He continued on his academic career in Cognitive Science (2008–2013), where he carried research on Brain - Computer parallelisms, as Masters Study. From 2013 on he is enrolled in Computer Engineering Ph.D. programme, focusing on Theoretical Computer Science, Operating Systems and Computer Networks. From 2016 on he specialises on Blockchain Ecosystems.

Can Özturan received his Ph.D. degree in computer science from Rensselaer Polytechnic Institute, Troy, NY, USA, in 1995. After working as a Postdoctoral Staff Scientist at the Institute for Computer Applications in Science, NASA Langley Research Center, he joined the Department of Computer Engineering, Bogazici University in Istanbul, Turkey, as a faculty member in 1996. His research interests are blockchain technologies, parallel processing, scientific computing, resource management, graph algorithms and grid/cloud computing. He participated in SEEGRID2, SEEGRID-SCI, and 1IP, 2IP, and 3IP European FP7 infrastructure projects and is currently participating in Infinitech Horizon 2020 project.