

# An Adaptive Load-Balancing Approach for Sharded Blockchains

David Halim Daou<sup>1</sup> and Ramzi A. Haraty<sup>1</sup>

<sup>1</sup>Department of Computer Science and Mathematics, Lebanese American University, Beirut, Lebanon

Received 4 Mar. 2024, Revised 28 May 2024, Accepted 6 Jun. 2024, Published 19 Sep. 2024

**Abstract:** The emergence of blockchains as a way to organize and protect user data across the internet has come with some concerns, mainly how to deal with the issue of scalability while still maintaining the security standards as well as the decentralized nature inherent to blockchains. Many different implementations will be elicited in this work, resort to the concept of sharding, consisting of partitioning a blockchain into smaller clusters which would, with proper load-balancing, drastically improve the performance of the overall chain. First, we will explore the existing literature on the subject as well as related algorithms. Then provide a detailed explanation of the functioning of the existing algorithms, both centralized and decentralized, as well as the one proposed in this paper. Next, we will then elicit the settings and conditions of the simulation environment, both in terms of data collection and preparation. Finally, we will provide the results obtained as well as a comparative analysis of the tested algorithms and give an overview of possible future endeavors in the advancement of load-balancing algorithms in respect to sharded Blockchains.

**Keywords:** Blockchain, Sharding, Consensus Protocol, Load Balancing, Centralized Algorithms, and Decentralized Algorithms

## 1. INTRODUCTION

Blockchains were introduced in 2008 [1] as a new way to store and process transactions in a secure and decentralized manner. They consist of storing transactions in a chain of blocks where each block contains the hash of the previous block. To ensure validity of a block, a consensus algorithm called Proof-of-Work must be executed by all nodes in the network that have the role of “miners”. This algorithm consists of pitting the different miners against each other where a computationally expensive puzzle has to be solved, each miner will have to provide a candidate block containing new incoming transactions and only one block will be chosen based on whether the majority of nodes in the network agree with the validity of the block and the transactions it contains. This ensures that as long as more than half of the network population consists of honest nodes, the resulting chain will maintain its correctness. Through the compensation of the “winning” miner in digital currency, nodes are incentivized/encouraged to participate in the Proof-of-Work algorithm. As such, the larger a blockchain becomes, the harder it becomes to compromise its security; thus, bypassing the single point of failure financial institutions can be vulnerable to. Figure 1 briefly illustrates the process of adding a new block to the blockchain.

Although the concept of blockchains started with the goal of supporting the Bitcoin network, new blockchains such as Ethereum [2] in 2013 would emerge, taking advantage of the decentralized and secure manner of the concept to build new cryptocurrency platforms as well as other applications such as healthcare.

Even though blockchains have significant advantages in terms of security thanks to their decentralized nature,

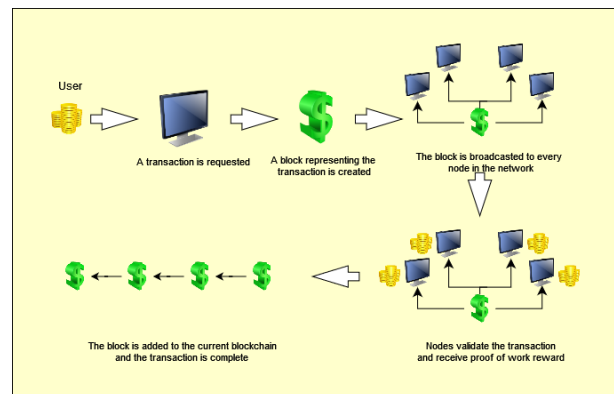


Figure 1. Blockchain Overview

they suffer from scalability issues [3] and are outperformed in terms of throughput per second for read and write operations as well as data storage volume by other systems such as Visa and PayPal. The fact that Proof-of-Work is extremely computationally expensive and lacks efficiency contributes to this slowdown. As such other 2 consensus algorithms were proposed to alleviate the negative effects of using the blockchain concept that started with the Bitcoin network.

### A. Blockchain consensus algorithms

#### Proof of stake:

The Ethereum network introduced a new consensus algorithm called Proof-of-Stake in 2022 [4] to improve the energy efficiency and performance of the blockchain. It relies on a staking mechanism to ensure that dishonest validators are penalized for potential fraudulent behavior

thus discouraging this type of behavior. A block is then proposed by a randomly selected validator. The validity of the proposed block is then verified by all other validators participating in the creation of the next block. This system not only helps securing the system through further decentralization and easier participation for nodes but also significantly increases throughput improving upon the scalability and efficiency of the blockchain. In Ethereum, a currency used called “gas” is expended on each transaction, the price and quantity required depends on the computational complexity of the transaction with a floor minimum of 21000 gas [5]. Validator nodes that are responsible for processing the transactions into blocks are then rewarded with a share of the transaction fees and distribute some of those fees to all other participating nodes in the staking process. To discourage malicious intent and enforce efficiency, validators that provide invalid transactions or who happen to be offline after their election as validator receive a penalty to their reward as a percentage of the staked amount.

### Practical byzantine fault tolerance:

The Practical Byzantine Fault-Tolerance consensus algorithm [6] relies on a voting protocol consisting of designated primary “leader” nodes that provide a block which must be validated through the voting of other nodes. This is achieved through 3 phases. If at least two thirds of the voter nodes agree on the validity of a block, then it is committed; thus, allowing a margin of failure based on majority voting. The significant multicasting and communications required result in significant communication overhead which itself results in bad scalability despite the increased energy efficiency. As such the Practical Byzantine Fault Tolerance algorithm is used in conjunction with Proof-of-Work in blockchain networks such as Ziliqa.

### B. Sharding

To improve upon the scalability of blockchain systems, some networks such as Ethereum [7], Chainspace [8], RapidChain [9] and OmniLedger [10] are considering the concept of sharding in future releases [11]. Sharding consists of partitioning a database into smaller regions that operate on local nodes. In the case of blockchains studied in this work, this consists of partitioning the entire network of account nodes into smaller groups as shown in Figure 2. Each group would then independently achieve consensus. This would ideally allow parallel execution and processing of new transactions; thus, significantly improving the performance of the network.

Despite its apparent advantages, sharding can suffer from issues related to the algorithm with which it is partitioned [12]. Improper balancing of computational loads among the different shards can lead to a significant decrease in performance despite the addition of parallelism. In Ethereum, transactions have varying computational complexities denoted by their gas expenditure [5], as such the gas spent by each successful transaction can be used as a means to calculate the complexity of each outgoing transaction per account; thus, denoting the total computational expenditure of each account in the

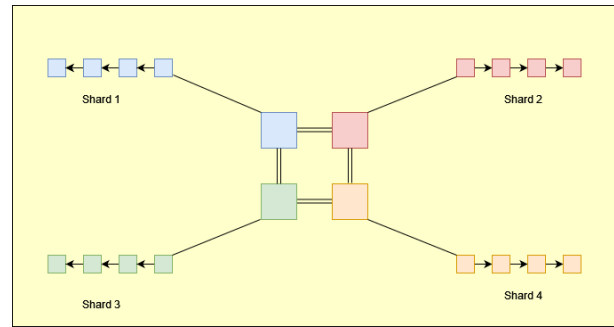


Figure 2. Blockchain Sharding

blockchain.

Another issue that arises from sharding is the increased vulnerability to 51 % attacks within the shard which can, in the worst-case scenario, result in an entire shard takeover compromising the entire network because of the reduction in the number of validators within each shard. As such, not only should the computational complexity be balanced among shards, but it should also be ensured that the number of accounts per shard is as well to minimize risk of a shard being compromised.

### C. Problem statement

The problem we will be addressing in this work is to design and implement an improved version of the diffusion load-balancing algorithms by tuning different parameters and provide a comparative analysis of the proposed algorithm, basic load-balancing algorithm and the regular diffusion algorithm proposed in the paper by Toulouse et al. [13]. We will be using data from Ethereum crawled with the online tool BitQuery Builder [14], and using the gas consumption of the transactions as a measure of load expended by each transaction.

### D. Contributions of the work

In this work, we provide an implementation of the distributed diffusion algorithm as well as an algorithm to handle account migration. Both algorithms have parameters and heuristics introduced to achieve better performance. We will then be providing a comparative analysis of the proposed algorithm variant and the unmodified one. We will also show how modifying tuning parameters as well as the introduced heuristics affect the performance of the proposed algorithms. To summarize our contributions:

- a) Adaptive diffusion algorithm implementation with both old and new topology.
- b) Optimized migration algorithm implementation.
- c) Experimental results and analysis.

The rest of the paper is organized as follows: Section 2 provides an overview of the load-balancing algorithms. Section 3 presents our model. Section 4 discusses our experimental results, and section 5 offers the conclusion and future work.

## 2. LITERATURE REVIEW

In this section, we will be reviewing the literature based on three criteria related to the proposed

model. First, we present load-balancing algorithms used in multiple different contexts, such as multiprocessor task scheduling or application client traces as well as sharded databases. This will mainly provide an explanation for the different and most common types of algorithms used in the simulation for performance comparison, specifically the centralized algorithms as defined later. Then, we will document the literature focusing on the specific case of load-balancing sharded blockchains though in different approaches than the one we will be using giving additional context on specific issues and implementations that could be considered. The final part of the section focuses on the literature regarding the specific type of algorithm we will be providing in the model. This will allow us to understand how these algorithms function, how fast they converge, which conditions make convergence possible and which prevent it, and what advantages and disadvantages do they have in terms of performance both in terms of efficiency of the balancing and the time complexity of the process.

#### Load-balancing algorithms

Load-balancing algorithms are responsible for distributing the loads among different servers so as not to overload specific servers and avoid creating bottlenecks which result in hampered performance. In our specific case, the load must be balanced among the different shards, so no single shard has a significantly larger load on its accounts than the other shards, both in terms of computational complexity and number of accounts.

There are two types of load-balancing algorithms, static and dynamic [13]. Static algorithms assume that all account data is known and assigns them to the corresponding shards before run-time. Dynamic algorithms, on the other hand take into consideration the current state of the network and assign the nodes to the shards based on this state. A hybrid approach of those types would be to query the current state periodically and apply load-balancing when needed.

Another observation to be made is the choice of whether all shards should be participating in the load-balancing process or only one, with the advantage of the former being decentralization and increased security by eliminating the possible existence of a single point of failure in the system. We will refer to these categories as centralized and decentralized load-balancing algorithms.

Another distinction to consider: whereas some algorithms automatically assign specific accounts to shards, other algorithms, as well as the one that will be proposed, divide the process into two steps. The first is to calculate the amount of load that must be transferred from one shard to the other. The second is to select the accounts from the pool of accounts available to a shard to transfer to neighboring shards.

We will next present the most relevant algorithms used in the literature for load-balancing, irrespective of the context.

#### A. Common load-balancing algorithms

Common static algorithms used for load-balancing include Round-Robin, First-Come-First-Served and Shortest-Load-First [15]. The main issue with those algorithms is their dependency on prediction of loads beforehand and as such are unsuitable for the case of blockchains. This applies to most static load-balancing algorithms and as such, unless they are called periodically over lapses of time, they would not perform proper load-balancing.

#### B. Longest processing time (LPT)

The Longest Processing Time algorithm [16] is an approximation algorithm that was shown to be optimal under certain assumptions on the input. It consists of sorting all loads in decreasing order while keeping track of the current state of the recipient shards, it then iterates through the loads assigning them to the least loaded recipients. It manages to do so with a worst-case running time of  $O(n \log(n))$  since it also requires sorting which uses the bulk of the algorithm's running time.

#### C. MULTIFIT algorithm

The MULTIFIT algorithm [17] was initially developed for identical multiprocessors task partitioning but can also be used to balance loads on a network of servers and thus on the sharding problem considered in this paper. It partitions the set of values in  $n$  subsets such as the largest subset sum is as small as possible and does so by using the First-Fit-Decreasing (FFD) bin-packing algorithm repeatedly until it finds the smallest capacity for the bin-packing to fit into the  $n$  required subsets. Similarly, to the Longest Processing Time algorithm its execution would have to be done on a single shard. It has a worst-case running time of  $O(n \log(n))$ .

#### Review of load-balancing in the context of sharded blockchains:

Various algorithms have been proposed in the literature that target specifically the load-balancing of sharded blockchains, which we will present next. We will then finish this chapter by focusing on the literature regarding the different type of load-balancing algorithms that we will be using.

In the work by A. Mizrahi and Rottenstreich [18], an approach is proposed that relies on clustering accounts based on the space requirements of each shard, selecting only the accounts with most frequent transactions for the clustering process. Then based on the mapping, the accounts are migrated to the corresponding shard. Although better performance in transaction processing time is obtained, the authors underline the issue of balancing both the load and the number of accounts per shard which can conflict with each other in the case of skewed data where some accounts are responsible for most of the transactions and thus take a larger chunk of load in each shard, posing a risk to shard security by having a smaller shard account population.

In their paper, Sangyeon Woo et al. [19], a heuristic is used that depends on the fact that the quantity of gas

expended in an Ethereum transaction represents its computational complexity and thus can be used to balance loads instead of the number of transactions per account. At the time of writing, it was shown to improve upon the existing account relocation mechanisms such as S-ACC and D-TX in terms of throughput and make-span.

In a subsequent publication authored by the same individuals, a dynamic load-balancing algorithm is employed, leveraging the previously mentioned heuristic of utilizing gas consumption as a gauge of workload [20]. Given the dynamic nature of this load-balancing algorithm, it depends on forecasting transaction workloads by accounts employing an aging weight that diminishes over time as an account remains inactive in initiating transactions. Subsequently, the algorithm is periodically invoked to ascertain requisite re-distributions. Superior performance metrics were achieved in both transaction throughput and make-span compared to the S-ACC algorithm, exhibiting an average enhancement of 9%.

In the paper by Li et al. [21], a prediction algorithm using Long Short-Term Memory (LSTM) is used to determine the load of incoming transactions in each shard as well as each account, considering only the accounts with most frequent transactions while grouping remaining ones into aggregate accounts, due to the lack of feasibility of doing so on every account. In a second step, the accounts are relocated with the goal of minimizing the variance between the loads of each shard. The accounts are sorted in decreasing order of the predicted loads thus moving from the most prolific nodes to the least, migrating the accounts while the variance obtained from doing so is smaller.

The work of Okanami et al. [22], propose a novel approach which consists of integrating a load balancing element in the validation process of the consensus algorithm of the blockchain. It does so by hosting a competitive game which focuses on solving an optimization load-balancing problem where accounts are the participants. To encourage participation, a reward is added to the account that submitted the best solution and to dissuade malicious behavior a fee must be paid prior to participation thus punishing the malicious players who would provide an imbalanced solution. This approach has been shown to significantly mitigate the consequences of selfish migrations which occur when a user decides to move to a less loaded shard for better throughput, resulting in widespread imbalance over the different shards hampering its overall performance. The selfish user would then be better compensated by participating in the competition rather than self-migrating to another shard.

In Krol et al. [23], the authors implement a prototype that provides account migration from one shard to the other and adds a recommendation system for these migrations. It does so by assigning to each account an alignment vector, which shows how strongly a vector is aligned to a specific shard, this vector as well as the vector of the receiving address of the transaction are increased by the cost of the transaction. The alignment vectors of a shard's accounts are then provided to all miners located

on this shard. Those vectors are only maintained for an interval of 100 blocks to the execution and only include the most active accounts. The prototype then selects a shard to which migrations will be made, opting for the least loaded shard with an account involved in the current transaction, in the case of new accounts choosing the least loaded shard in the network. Migration of an account is then decided by comparing the product of the account and the cost of the transaction with the sum of this accounts' alignment with other shards, if the product is smaller than this sum then the account is migrated to the designated main shard. The prototype was shown through testing to improve upon overall transaction throughput of the blockchain while remaining comparatively lightweight due to its integration with existing protocols used by miners. To financially incentivize and encourage miners, a reward mechanism is added rewarding miners for maximizing throughput.

In [24], the authors attempt to solve the load-balancing problem in sharded blockchain with the goal of minimizing cross-shard transactions, which result in significant overhead, by periodically executing two different allocation algorithms with different frequencies. The former optimizes the account allocation based on all historical account data whereas the latter is executed more frequently and executes allocation based on the former's results focusing only on incoming transactions. The first algorithm turns the optimization problem into the community detection problem and uses the Louvain method to solve it in  $O(nk+n \log)$  time complexity. The second is an adaptive algorithm, that instead of considering the entire historical data, only considers the last account to shard mapping and the incoming transaction, thus executing faster than the former with constant  $O(1)$  time complexity. Through simulation, the authors showed that the number of cross-shard transactions was reduced from 98% to 12% thus improving overall throughput.

Review of distribution algorithms in the context of load-balancing

Distribution algorithms were first introduced by Cybenko [25] in 1989 as a way to perform dynamic load balancing among multiprocessors. Each processor is represented by a node in a connected graph so that every 2 processors are shown to significantly mitigate the consequences of self-linked, the edges between each processor are stored in an adjacency matrix of the network. It consists of using the following formula to calculate load transfers that must be made from one processor to its neighbors:

$$w_i^{t+1} = w_i^t + \sum_j^N ij(w_j^t - w_i^t) \quad (1)$$

The variable  $w_i^t$  represents the workload of processor  $i$  at epoch  $t$ , the summation iterates over every neighbor of the currently examined processor and  $ij$  represents a weight matrix fulfilling the condition that  $ij = 0$  only if the 2 processors are not connected to each other. Should  $w_i^{t+1}$  be positive then processor  $i$  should transfer that specific workload  $w_i^{t+1}$  to processor  $j$ , otherwise it is

processor  $j$  that should send this workload to processor  $i$  to achieve balance between the 2. This algorithm is decentralized by nature as each processor is responsible for its own load transfers. Repeatedly applying the algorithm should eventually lead to convergence though the loads are not in niteely divisible. This type of algorithms would be shown to be stable by Berenbrink et al., [26], thus the total load of the system is bounded over time with an asymptotically tight upper bound. The main condition for this stability is that the graph must be connected.

The problem di usion algorithms try to solve is known as the Distributed Averaging Consensus problem which is mainly relevant to large-scale decentralized networks such as blockchains. The following two papers by Xiao et al. [27] [28] focus on the selection of the weight matrix used by the algorithm so as to ensure convergence.

The rst paper proposes the use of the Metropolis-Hastings matrix defined in the following formula:

$$W_{ij}(t) = \begin{cases} \frac{1}{(1 + \max\{d_i(t); d_j(t)\})} & \text{if } i, j \text{ are neighbors} \\ W_{ik}(t) & \text{if } i = j \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

The weight of each edge is 1 over the max degree of the 2 vertices, the matrix is therefore stochastic as the sum of each row is equal to 1 and the condition specified earlier,  $w_{ij} = 0$ , is respected as the weight at index  $ij$  is equal to 0 when  $i$  and  $j$  are not connected by an edge. In the second paper by Xiao et al., they propose solving the least-mean-square consensus problem to determine the weight matrix's values instead of the Metropolis-Hastings matrix. The Metropolis-Hastings matrix was shown to have slower convergence than the resulting matrix on average.

To solve the step problem that can occur in di usion algorithms, [29] Jeannot & Vernier propose an approach using simulated annealing which can detect whether the algorithm would converge thus eliminating the step problem. The convergence detection algorithm was shown to provide, in the best case, a 100% performance increase while not degrading the di usion algorithm's performance.

The paper by [31] explores the conditions surrounding nite-time consensus in averaging algorithms by comparing it with consensus in maximization and minimization algorithms. The graph considered for the analysis has a time-varying topology, which means that some edges might be added or removed, similar to a network losing its connection to a node because of external events. They then demonstrate that there exists necessary conditions for reaching a global nite-time consensus. The rst is that the initial values of each node belong to a set of integers they call the consensus manifold which is one-dimensional. This means that reaching a global nite-time consensus is almost impossible. The second condition is that the graph is quasi-strongly connected, which means that the graph contains a node that is linked that can reach every other node in the network.

In [32], the authors attempt to determine the sufficient and necessary conditions for convergence of average consensus algorithms using a Metropolis-Hastings weight matrix. Instead of arbitrarily adding one to the denominator, the variable is summed with the maximum degree of both nodes. By modifying this variable, they identify both sufficient and necessary conditions for convergence on various topologies. They show that  $0$  is a sufficient condition for convergence regardless of the topology. They also show that convergence is guaranteed with  $0$  assuming that the graph is not regular and bipartite. In the case of regular and bipartite graphs, they show that the probability of convergence decreases exponentially as the total size of the graph increases. They also show that the optimum is roughly reached for these topologies with inversely proportional to the number of nodes in the network.

$$W_{ij}(t) = \begin{cases} \frac{1}{\max\{d_i(t); d_j(t)\} + 1} & \text{if } i, j \geq 2 \\ W_{ik}(t) & \text{if } i, j < 2 \text{ and } k, i \\ 0 & \text{otherwise} \end{cases} \quad (3)$$

In [30] Lopes et al., provide an analysis is provided for di usion algorithms on distributed networks, mainly considered to tackle the load-balancing problem of sharded blockchains. They define the problem as such: the performance of distributed averaging estimation-work is represented by an un-directed graph where each node corresponds to a vertex and edges correspond to direct connections between graphs. The network topology considered is a ring network, a regular graph, where each vertex has exactly 2 neighbors. Regular graphs are then fused with an adaptive network layer that detects changes in the neighborhood of each combiner and updating the values of the combiners accordingly. They

is used as the weight matrix with which the calculation of required load transfers is made, in the case of a ring network, it can be determined that each row would contain exactly 3 entries that are not 0, those entries would be equal to  $\frac{1}{3}$  on the condition that  $a_{ij} = 1$  or  $i = j$ . As such the matrix obtained is row-stochastic and fulfills the condition that a weight is equal to 0 only if both vertices are not connected. They provide the following pseudo code that is executed in parallel by each shard:

---

Algorithm 1: Diffusion algorithm by [33]

---

Input:  $i$ ,  $N_i$ ,  $n$ ,  $W$ , workload of shard  $i$

```

int Loadi(0) = workload of shard i
int t = 0
float  $\Delta_i[n] = 0$ 
while no convergence do
  foreach Neighbor of  $i$  do
    |  $\Delta_i[j](t+1) = \Delta_i[j](t) + W_{ij}(\text{load}_i(t) - \text{load}_j(t))$ 
  end
   $\text{load}_i(t+1) = \text{load}_i(t) + \sum_{j \in N_i} \Delta_i[j](t)$ 
  t = t + 1
end

```

---

Figure 3. Ring network topology example

The variable  $\Delta_{ij}$  represents the transfer vector of shard  $i$ ,  $N_i$  represents the number of neighbors of  $i$ ,  $W$  represents the weight matrix,  $t$  represents number of iterations in which the transfer vector is filled and the resulting load from the transfer calculated. The authors assume convergence is reached when the sum of all initial loads is divided by the number of shards, so the total load of the system is over the number of shards. Positive entries in  $\Delta_{ij}$  mean that shard  $i$  should transfer a load of size  $\Delta_{ij}$  to neighbor  $j$ , negative ones mean that shard  $i$  should instead be receiving this load amount from neighbor  $j$ . The authors then test the algorithm and show that the approach results in similar output to other load-balancing algorithms such as LPT and MULTIFIT while being faster, though they note several shortcomings such as the heuristic used to select and migrate the account nodes that can result in different values than the ones computed in the diffusion algorithm, they also note that the topology of the network may negatively affect the migration process as shards can only migrate to neighboring ones in the topology.

In [13], the authors try to improve upon the last attempt, mainly by using actual blockchain data rather than artificially generated datasets for the experiments, and propose another version of the diffusion algorithm that computes the loads in an optimal number of iterations. They use a migration algorithm based on a heuristic that consists of sorting in decreasing order the account balances on each shard and migrating every shard that is smaller or equal to the transfer vector. The implementation is dynamic in the sense that it attempts load prediction to add new transactions in real time using long short-term memory (LSTM) neural networks, instead of periodically redistributing the accounts among the shards. The network topologies used in the experiments are 2 regular ring networks, torus networks and line networks. The results obtained show that both algorithms

are not successful in the improvement of the load-balancing among the shards, the authors suggest that the prediction of incoming loads is to blame since newly created accounts cannot have their loads predicted and suggest a periodic repositioning combined with short-term positioning of new accounts could prove more successful.

### 3. Model Description

In this section, we will first be providing a comprehensive description of the proposed model and how the algorithms used are implemented, and providing explanations for some decisions in its design, specifically the numerous parameters added to regular diffusion algorithms in order to facilitate convergence and improve performance. Then in a second part, we will be presenting a use case in which the model is applied to an imbalanced sharded blockchain and walk through the steps required to apply it. It is worth noting that the model in question assumes that account location and migration within the shards are not in the control of individual nodes, as such, malicious behavior may mainly manifest from extremely rare cases where more than 51% of a shard's load belongs to a group of colluding malicious accounts which could potentially take over the shard. Figure 4 shows the example of how a 51% attack is perpetrated. This decision also prevents selfish behavior from account nodes attempting to move to other shards for better throughput and rewards. Due to this assumption, it is fair to assume that potential collusion would have to take place after colluding nodes have both been added to the same shard and not be premeditated or spontaneously provoked by voluntary migration. Also, account nodes which would attempt shard takeover would have to disburse significantly large amounts of currency and thus the approach would not be viable nor lucrative to attempt.

Another assumption is that the model is expected to be called periodically, the addition of new transactions and accounts is not implemented, though we assume that newly created accounts are added to the shard with the least number of nodes. This is to ensure that when the model is used, the resulting model is as balanced as

always guaranteed on the global average of the network. The standard deviation is therefore a good measure of proximity to the average of the network.

Figure 4. 51% attack example

possible in terms of number of nodes per shard, and thus to avoid the possibility that some shards are much less populous than others increasing risks of a 51% attack from malicious accounts.

A. Model overview

The model will first pre-process Ethereum transaction data extracting the address of the sender and the gas consumed by a transaction. The transactions will then be grouped per their account address. This will then give us the account balance of each account and the total load an account has on the network. We then normalize the gas values by dividing them by 21000 which is the minimum amount of gas that must be provided by an Ethereum transaction.

We then decide upon the number of shards to be used to partition the network. Since each shard is expected to be relatively large, we can assume that all shards are directly connected to each other, so we can assume that the graph is  $(\frac{1}{n})$  regular for  $n$  shards. The reason for this assumption is that it only takes one account of a shard to be reachable from any one node from another shard for both shards to be considered connected. In the case of large, decentralized peer-to-peer networks such as Ethereum, this assumption is evident, and the possibility that a shard is disconnected from any other shard in the blockchain is extremely improbable. This topology is therefore a more accurate representation of load-balancing sharded blockchains such as Ethereum. A possible downside of that approach is the overhead that would result from making load access for transfer vectors mutually exclusive, possibly slowing down the algorithm for the sake of consistency and correctness.

To properly simulate and demonstrate the effectiveness of the algorithm, the accounts are randomly split into shards of equal account capacity. As such, the shards are balanced account-wise but since not all accounts have the same load, the shards obtained are not necessarily balanced. This provides us with a good way to test the enhanced diffusion algorithm.

The pseudo code defined earlier by [33] (Algorithm 1) is used, though to establish convergence, we make use of the standard deviation of the shard distribution as an additional tuning parameter. The reason for this decision is because the amount of data and the dependence of diffusion algorithms on the initial loads, convergence is not

Figure 5. Example of a 3-regular graph (4 nodes)

We use the Generalized Metropolis-Hastings matrix to obtain the weights, to do so we take the adjacency matrix of the network and compute the weights accordingly. To maximize the probability of convergence we set  $\frac{1}{n}$ , where  $n$  is the number of shards as was shown in [32] to be roughly equal to the optimum needed to guarantee convergence in regular graphs of larger size.

$$W_{ij} = \begin{cases} \frac{1}{\frac{1}{n} + \max(d_i(t); d_j(t))} & \text{if } i, j \text{ are neighbors} \\ W_{ik} \frac{1}{n} & \text{if } i = j \text{ and } k \neq i \\ \frac{1}{n} & \text{if } i = j \text{ and } k = i \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

By doing so we allow each shard to transfer load directly to shards with a load too small regarding the threshold given by the distance of the standard deviation with the global average. This makes convergence faster and significantly more frequent than selecting the average, convergence in the average may not be possible.

We also added two shard selection algorithms, one checks which shards are overloaded and the other one which are under-loaded, by applying the convergence rule defined earlier. This means that the diffusion and migration 10 algorithms have to go through at least two epochs, one for the smaller loads and one for the larger ones. As such we added a parameter that handles the alternation between both types of epochs. To properly apply this parameter, we flip the value after each selection,



where a value of 0 would mean that larger loads must be selected whereas 1 would mean the smaller loads have to be selected. This decision was made to avoid unnecessary overhead caused by creating additional threads for shards that have already converged with the global average, it will also be beneficial to greatly decrease the migration process by only selecting the shards that have to send loads to neighboring ones.

We then call the diffusion algorithm on each selected shard in parallel and let their transfer vectors, thus designating how much load must be migrated from one shard to the other. Once the transfer vectors have been computed we can then start migrating nodes from overloaded shards to less loaded ones to achieve overall better balance. To do the migration, each shard must select candidate accounts for transfer, the heuristic used to do so is to sort each shard's accounts in decreasing order of load and selecting a node whenever it is smaller or equal to the transfer required, the node is then migrated and both shards updated. Contrarily to the migration algorithms, parallel diffusion is possible since the transfer vectors computed are stored in different parts of a greater list, each dedicated to its shard's required transfers, thus it is impossible for more than a single entity to read and write to the same transfer vector location in memory and causing a race condition in the process.

This approach to migration can be an issue, as in the case where the transfer is extremely small, the migration would take more time since it would have to iterate over every account, hampering overall performance. To alleviate this, we decided to add a tuning parameter, similar to the one used for convergence which is to place a margin of acceptable imbalance based on a fraction of the global average. Although this makes perfect balance impossible, it also avoids the significant decrease in performance required to achieve it on distributed systems and accelerates consensus while guaranteeing an upper bound of imbalance of our choice, which in extreme cases would revert to parameter-less migration.

We have also implemented a second migration algorithm for testing that uses a different heuristic. The heuristic used consists of sorting each shard's nodes in decreasing order of size, only instead of iterating over the entire shard, it fetches the first transaction with a load of the size of the transfer to be made. This proves useful in the case of smaller transfer loads as it avoids iterating through all nodes preceding it. We will also be comparing the performance both in terms of accuracy and time taken of both heuristics for migration and show the cases in which one would be preferable to the other.

Based on our handling of the simulation environment, migration had to be applied on a single instance, in an actual peer-to-peer scenario meaning that migration is either handled by an additional entity such as the main chain keeping track of the shards or a single shard would be designated to do the task and then broadcast to the other shards all necessary migrations. As such, the migration algorithm must delete a node from one shard as well as add it to another, which means that it could

cause a race condition in which 2 or more shards try to modify the same account at the same time and result in varying or possibly faulty node data. A helper function has also been added in case the number of designated epochs is more than necessary to reach load-balance. It consists of verifying before execution if all shards converged within a margin bounded by the distance to the initial standard deviation of the model that was defined before the first epoch. This parameter helps us regulate the total imbalance of the network while making sure that the algorithm finishes execution. The reason for this choice is that using the latest standard deviation could result in values between 0 and 1 which would be impossible to achieve and would prevent convergence.

#### B. Model use example

To show how the model works we will be presenting the case of a sharded Blockchain similar in its functioning to Ethereum. As such, there exists a resource expended on transactions proportional to the computational expenditure of the processing of the transaction in the network, such as the gas resource does in the Ethereum network. The Blockchain shards in question are also in need of balancing and the approach taken is account-based, which means that we are not only trying to balance the loads of the accounts but the number of accounts as well.

We decided to divide the network into 5 shards for this example. Figure 6 shows an example of the initial distribution obtained when dividing the overall blockchain network into 5 shards.

We assume that in its initial state, the sample Blockchain is sharded in such a way that every shard has approximately the same number of accounts, but accordingly, the total load of the accounts located on each shard is imbalanced above a certain pre-defined ratio. We decided to arbitrarily define the measure of tolerable imbalance as a fraction of the initial standard deviation instead of trying to eliminate all load imbalance. In this example we choose 0.1 of the initial standard deviation.

Figure 6. : Initial Load Distribution of the sample Blockchain

We then compute the Generalized Metropolis-Hastings matrix as defined earlier in [32] to get the weights of a  $(n-1)$  regular graph in which each vertex corresponds to a shard itself pointing to the accounts within it. We decided as per the recommendation of the



paper introducing it, to define  $\epsilon = \frac{1}{n} = \frac{1}{5}$ , this is to increase the chance of convergence of the algorithm.

The division and migration process will iterate over a finite number of epochs alternating between shards which have a higher load than the maximum tolerable one and those that have a smaller load than the minimum tolerable load, until the total imbalance of the network among all shards is within the tolerable threshold set by the initial standard deviation.

On each iteration (or epoch), we first check which specific shards need balancing and apply division to these shards in parallel. We also compute the current standard deviation which decreases on each iteration, this should allow us to reach more accurate results with each epoch until the fraction of the initial standard deviation is reached. This fraction can be modified based on the minimum imbalance tolerated by the Blockchain in regard to its shards.

The division algorithm will then attempt to calculate the transfer vector of the current shard, to determine how much to send to the other shards. To avoid sending accounts to already overloaded shards, the shards are sorted from the lowest loaded to the highest loaded in case the current shard is overloaded and from the highest loaded to the lowest loaded in case the current shard is under-loaded. This optimization is used to ensure that the neediest shards are selected first on each iteration of the loop. The transfer vector values from the current shard to the other ones are computed by applying the previously defined weights to the current loads of the concerned shards, the currently examined neighbor and the shard itself.

Once the current shard is shown to converge, the loop ends, and the final transfer vector is computed. The next step will consist of converting the negative values within the vectors to positive ones for the corresponding shards. As such migration will be able to focus on sending rather than implementing both sending and receiving tasks.

Figure 7. Final load distribution of the sample Blockchain.

In the final step, we iterate over every shard and apply the migration procedure, transferring excess loads to less loaded shards until the transfer vectors of each shard consists only of zeroes. Thus, on each successive iteration

of the division/migration process, the total imbalance of the network decreases until it reaches the tolerated threshold. In the specific example given, the process resulted in reaching a total imbalance of 1346 among the different shards as opposed to the initial imbalance of 26502. Figure 7 shows the obtained distribution of loads for the example case given above. The (1) regular topology also proved to be faster than the regular topology, both in terms of number of iterations and running time (37 seconds vs. 25 seconds, 7 epochs vs 20 epochs).

Since the algorithm would be called periodically and the new transactions would follow the heuristic mentioned earlier to be committed to the ledger, as such overall delay on the Blockchain transaction throughput is heavily dependent on the new transaction sizes added between each period, assuming the acceptable threshold of imbalance is kept for a period then the delay would be minimal, since no migration or execution of the division algorithm would be required, and the only required operation would be to compute and compare the loads of each shards which would simply mean going over each account load in each shard and computing the sum for each shard. This operation would result in an O(n) time complexity where n is the total number of accounts in the Blockchain at the time of execution, though caching could be used to minimize this delay by storing and updating the shard loads as transactions are added.

In the worst case, both division and migration would be required which due to the unpredictability of distributed algorithms, such as the division algorithm, may vary and also is heavily dependent on the error metric used to establish convergence. The choice of a smaller fraction of the standard deviation as acceptable error would increase the delay of the division algorithm. The selection process of the shards that would be required to meet on the time complexity. As such the overall time complexity of the division process would be  $O((N - 1) \log_{\frac{1}{E_1}} \log_{\frac{1}{E_2}} c)$  where E1 is the inner error threshold used for local convergence, E2 is the error threshold used for global convergence and c is the total number of selected shards per iteration. Due to the nature of the algorithm using parallelism, the provided complexity can also be applied to the algorithm without parallelism although execution time would most likely be lower due to the linear execution of the algorithm.

The migration algorithm execution however is dependent on the results of the division algorithm, ranging from no migrations to significantly more depending on the required transfers as well as the loads of each individual account from the sending shard, assuming hashing is used, as is the case with the proposed model, finding the candidate accounts to transfer can be done in constant time. As such the overall complexity would be O(c) where c is the number of accounts that were selected which due to the greedy nature of the account selection is minimized.

4. Experimental Results

In this section, we will start by eliciting the software used in the simulation. Then we will move on to how the

testing data was gathered and processed for use in the experiment, as well as underline the different algorithms and settings used in the comparative analysis. We will then finalize the chapter by showing the obtained results using bar graphs.

#### A. Simulation software used

The Jupyter Notebook web interface was used to conduct the simulation and tests in Python 3. The choice of Python was based on the assumption that the algorithm showcased would require sufficient data processing capabilities and quality of life features to reliably be tested and compared. Since Python libraries such as Pandas and Numpy offer such functionalities, it proved a better choice than taking advantage of the Object-Oriented capabilities of Java which was initially chosen for the simulation.

#### B. Data preparation

To reliably conduct the simulation, data was obtained through two different methods. The first consists of crawling Etherscan [34] which is a platform used to explore Ethereum blocks and transactions. To simplify the crawling process, BitQuery Builder [14] was used to make the API calls and extract the transactions within specific intervals. The crawling was therefore done three times each underlining the first 2500 transactions to be processed within each month of a three-month period. The second method was to obtain static data from Kaggle [35] containing 100000 last transactions, the main advantage of this method is the size of the obtained dataset as BitQuery does not allow retrieval of more than 250 transactions per call.

After obtaining the data, we filter out transaction data irrelevant to the simulation and keep the gas expenses and address of the sender of each transaction. We then group the total gas expenses per account address. Any outstanding account with more than 20 ether of total gas expenses is removed to avoid bias and for experimentation's sake.

To further simplify the data presented, we decided to divide all gas expenses by the minimum amount of gas expenses allowed which is 2100 gwei; thus, 000021 ether, so that accounts that only did a single transaction would have a gas expenditure of 1. This will be useful to improve upon the performance of the second migration algorithm proposed.

#### C. Algorithms and simulation metrics

The algorithms used and implemented for comparison are Longest Processing Time, regular ring network diffusion and (1)-regular network diffusion. In addition, to properly conduct comparisons, both migration algorithms are tested and included in both diffusion algorithms. We will be testing each algorithm on 5, 6, 7, 8, 9 and 10 shards on all four datasets. The parameters used for convergence will be set to 20% of the initial standard deviation of the tested allocation for cases where the number of shards is equal to or higher than 8 shards and 0:1 for strictly lower than eight. We take this measure to maximize chances for convergence. The metrics used to determine the performance are the total load imbalance among shards multiplied by the execution time of the process. A lower product would therefore correspond

Figure 8. Dataset 1 table

The results for each dataset will be shown separately in their own graph, the x-axis will represent the number of shards used in the test and the y-axis will represent the total load imbalance for this specific shard on both algorithms. Therefore, we will be able to compare side by side the overall performance of both algorithms on a given number of shards, we will also be providing dataset characteristics such as the total number of accounts, the total load of accounts as well as the ideal global average for this shard configuration. It should be taken into consideration that there is a significant dependency of the distribution of accounts on each shard. Therefore, results might be inconsistent between different shard numbers, though we will be underlining major trends across the tests by focusing on each algorithm's performance on the same initial distribution. For the purpose of this experiment, we conducted and recorded a total of 48 test configurations.

#### D. Results

In this section we will show the results of the simulations each dataset at a time. Each subsection will specify the characteristics of the examined dataset, eliciting the dataset size, both in number of accounts and load and the ideal average for the specified shard size. Then we will be providing a table for each dataset showing the results obtained for each shard configuration with each algorithm. A line diagram will be given to further visualize the results and check for specific patterns across the different datasets.

##### 1) Dataset 1

The first dataset contains data from 48857 Ethereum accounts with a total load of 718312 across all accounts. The ideal average load for shard counts of 5, 6, 7, 8, 9 and 10 are respectively, 143662, 119719, 102616, 99789, 79812 and 71831. The largest transaction in the dataset expends 20000000 gas and the smallest ones expend 21000 gas since it is the minimum allowed for a transaction in Ethereum.

As can be seen the (1)-regular diffusion algorithm outperforms the 2-regular one on all shard sizes in term of total execution time. It also manages to get a better score on every shard number aside from a size of 5 shards. It

is worth noting that although better score and execution work. For the purpose of this study it is worth noting that times are obtained using (1)-regular division, we score for (1)-regular division grows at a numerically obtain increased imbalance in terms of account numbers slower rate than 2-regular division (2471 and 9686 for in every shard size. We also notice that some shard size of 10 shards). Account number imbalance though is numbers obtained considerably worse scores on certain much higher on the (1)-regular model. We theorize this shard configurations with both algorithms, though that could possibly be due to the small dataset and significant might be due to the initial distribution of the accounts inequalities in account loads, as additional accounts might among the shards and the fact that perfect account numbers have to be displaced to compensate for the most loaded balance might not be reachable for every shard sizes resulting in multiple accounts being moved between which results in the initial distribution to start slightly shards. overloaded or under-loaded in terms of account number.

Figure 9. Dataset 1 diagram

Figure 11. Dataset 2 diagram

2) Dataset 2

3) Dataset 3

The second dataset used consists of 9407 accounts with a total load of 350924. The ideal target load across the shards for each number of shards are 70185, 58487, 50132, 43866, 38992 and 35092, for 5, 6, 7, 8, 9 and 10 shards respectively. The largest transaction in this dataset uses 703797773 gas.

The third dataset consists of 24984 accounts with a total load of 821224, the ideal load to reach for sizes of 5, 6, 7, 8, 9 and 10 shards are respectively, 164245, 50132, 43866, 38992 and 35092, for 5, 6, 7, 8, 9 and 10 shards respectively. The largest transaction in this dataset uses 1846089111 gas.

Figure 10. Dataset 2 table

Figure 12. Dataset 3 table

We can make the following observations from this dataset. First, (1)-regular division performs better than 2-regular division on all tests in terms of execution time, on the other hand, account imbalance is higher with (n-1)-regular division due to the larger number of migrations enacted to reach the final loads. Another observation to be made, is that (1)-regular division performs better in terms of score though the significance of that improvement seems largely dependent on the initial distribution as well as the shard count used. In the above test, for example, on a shard count of 7 and the same initial distribution for both division algorithms, (1)-regular division performs only slightly better than 2-regular division with a score of 606 against a score of 649, while the account imbalance is much higher on the former than the latter. In this specific test, it can be safely assumed that 2-regular division is overall more

effective than  $(n-1)$ -regular. In other tests however, such margin of error. It would therefore seem that selecting as 8 and 9 shard tests,  $(n-1)$ -regular division performs round numbers by which any possible load is divisible significantly better than 2-regular division in terms of score while having a marginally worse account imbalance blockchain.

So far, with the observations made in this dataset, there seems to be a trade-off between using 2-regular division and  $(n-1)$ -regular division, in terms of whether the blockchain network in question would result in better balance in terms of load for better performance or better security through better balancing of individual accounts throughout the shards.

Figure 13. Dataset 3 diagram

#### 4) Dataset 4

The last dataset consists of 24988 accounts with a total computational load of 820239 among them. The ideal loads are respectively, 164048, 136707, 117177, 102530, 91138 and 82024, for 5, 6, 7, 8, 9 and 10 shards. The largest transaction in this dataset uses 1244473943 gas.

Figure 14. Dataset 4 table

Similarly, as on the previous tests, execution time is shorter throughout the tests on  $(n-1)$ -regular division. On the other hand, on the size of 5 shards, 2-regular division performs better in terms of score and account imbalance, though as the number of shards increases,  $(n-1)$ -regular division becomes preferable in terms of score culminating with a score of 5246 on  $(n-1)$ -regular division against a score of 31907 on 2-regular division for a shard number of 9. An important observation to be made is that a shard size 10 results in a better score and execution time than shard size 9 with both division algorithms, implying that the number of shards has a significant impact on the performance of division algorithms, mainly when the total load divided by the number of shards results in an infinite quotient which would only be attainable approximately due to rounding which would result in a

Figure 15. Dataset 4 diagram

#### E. Overall observations

Throughout the conducted experiments, some general observations can be made about the relative performance of our  $(n-1)$ -regular division algorithm with the 2-regular division that has been considered to do load-balancing on blockchains such as Ethereum. Firstly, the  $(n-1)$ -regular variant is consistently more effective in terms of execution time throughout all datasets and shard counts. Secondly, as the number of shards increases in the network, the improvements introduced by the  $(n-1)$  variant, both in terms of load-balancing and execution time, become much more significant, whereas 2-regular division performance greatly decreases. Lastly, in terms of account balancing, the improvement of the load-balancing comes at a higher cost. Due to the greater number of migrations caused by the topology where every shard is interlinked to all others, the account balance is significantly worse. Thus, there seems to be a trade-off between using both algorithms.

Another slight observation to be made is that of the specific number of shards on all division algorithms' performance, with some higher numbers performing significantly better than lower ones. We theorize that the number of shards a blockchain network is to be divided into should be carefully selected as not to cause rounding errors due to infinite quotients as ideal averages.

Since the algorithm would be called periodically and the new transactions would follow the heuristic mentioned earlier to be committed to the ledger, as such overall delay on the Blockchain transaction throughput is heavily dependent on the new transaction sizes added between each period, assuming the acceptable threshold of imbalance is kept for a period then the delay would be minimal, since no migration or execution of the division algorithm would be required, and the only required operation would be to compute and compare the loads of each shards which would simply mean going over each account load in each shard and computing the sum for each shard. This operation would result in an  $O(n)$  time complexity where  $n$  is the total number of accounts in the Blockchain at the time of execution. In the worst case, both division and migration would be required which due to the unpre-

dictability of distributed algorithms, such as the diffusion algorithm, may vary and also is heavily dependent on the error metric used to establish convergence. The choice of a smaller fraction of the standard deviation as acceptable error would increase the delay of the diffusion algorithm.

## 5. CONCLUSION AND FUTURE WORK

In this paper, we presented an alternative implementation and imagining of the load-balancing diffusion algorithms for sharded blockchains. The solution proposed retains the main advantage in its distributed, decentralized nature while improving upon the performance of the load-balancing procedure by considering a different network topology whereby all shards are connected to each other, to reduce the number of iterations the diffusion algorithm would need to execute the load-balancing.

By reducing the number of passes required to perform load-balancing using  $(n-1)$ -regular topology, we successfully managed to reduce the overall execution time of both the load estimation and node migration procedures reaching our initial goal. We have also obtained better performance as the number of shards grows, which would be beneficial in the case of larger networks which can accommodate more shards to leverage the advantages of parallelism and decentralization. We also manage to do so within acceptable boundaries through the use of total account number imbalance between the shards to ensure we minimize the risk of compromising shards that are account under-loaded. On a minimum of 5 shards, we obtain slightly better scores on our approach than the ring topology on the various datasets, though as the number of shards grows to 10, the improvement becomes significantly higher such as in Dataset 2 where we get a score of 2471 on the  $(n-1)$ -regular topology against a score of 9686 on the 2-regular.

Despite the apparent advancements, these enhancements entail potential consequences concerning account balance, and in certain extreme scenarios characterized by disparities in account workloads, they may render shards more susceptible to 51% attacks. We can assume that further increasing the number of shards on smaller datasets could potentially make the approach detrimental to the security of the individual shards and by extension the entire blockchain because of the account imbalance and the fact that some accounts contain much larger computational loads than others. We suggest future work on the matter of sharded blockchains should introduce mechanisms to detect such risky situations and to alleviate the effects by increasing the threshold under which a load imbalance is tolerable, thus sacrificing performance to maintain a dependable level of security. Another possible approach to this problem would be to introduce a tolerance threshold mechanism for the number of accounts per shards, this variable would then make sure that the minimum number of accounts over the shards stays above a certain value to guarantee the security of the shard. Upon breaching this threshold, the algorithm would then revert to its pre-diffusion state and increase the load threshold.

Another issue that should be addressed would be how well the  $(n-1)$ -regular topology can be applied to other

sharded blockchains than Ethereum such as Ziliqa or Hyperledger Fabric. For the sake of this paper, Ethereum was mainly chosen for accessibility of data on its transactions, such as the gas expenditures which in turn allows us to find total account computational loads. We posit that the blockchain staking mechanism would be essential to the functioning of such as method, so the Hyperledger Fabric consensus mechanism would have to be a Proof-of-Stake algorithm to apply the sharding approach proposed above since it provides a convenient and scalable way of representing computational loads per account.

Overall we managed to improve upon the scalability of a sharded Ethereum blockchain which is widely used and a staple among the Proof-of-Stake consensus blockchains. Though it remains to be seen how well it can be applied to smaller account-based Proof-of-Stake blockchains or even Ethereum itself if the number of shards was to be greatly increased, to a scale where 51% attacks would become feasible.

## REFERENCES

- [1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *Decentralized business review*, 2008.
- [2] Q. Huang, "Ethereum: Introduction, expectation, and implementation," *Highlights in Science, Engineering and Technology*, vol. 41, pp. 175–182, 03 2023.
- [3] A. I. Sanka and R. C. Cheung, "A systematic review of blockchain scalability: Issues, solutions, analysis and future research," *J. Netw. Comput. Appl.*, vol. 195, no. C, dec 2021. [Online]. Available: <https://doi.org/10.1016/j.jnca.2021.103232>
- [4] Github. (2024) Consensus-specs. [Online]. Available: <https://github.com/ethereum/consensus-specs>
- [5] Ethereum Foundation. Gas Fees. Ethereum Foundation. [Online]. Available: <https://ethereum.org/en/developers/docs/gas/>
- [6] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Trans. Comput. Syst.*, vol. 20, no. 4, p. 398–461, nov 2002. [Online]. Available: <https://doi.org/10.1145/571637.571640>
- [7] GitHub. (2024) Ethereum Wiki. [Online]. Available: <https://github.com/Ethereum/wiki/wiki/>
- [8] M. Al-Bassam, A. Sonnino, S. Bano, D. Hrycyszyn, and G. Danezis, "Chainspace: A sharded smart contracts platform," 2017.
- [9] M. Zamani, M. Movahedi, and M. Raykova, "Rapidchain: Scaling blockchain via full sharding," *Cryptology ePrint Archive*, Paper 2018/460, 2018, <https://eprint.iacr.org/2018/460>. [Online]. Available: <https://eprint.iacr.org/2018/460>
- [10] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford, "Omniledger: A secure, scale-out, decentralized ledger via sharding," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 583–598.
- [11] G. Yu, X. Wang, K. Yu, W. Ni, J. A. Zhang, and R. P. Liu, "Survey: Sharding in blockchains," *IEEE Access*, vol. 8, pp. 14 155–14 181, 2020.
- [12] E. Fynn and F. Pedone, "Challenges and pitfalls of partitioning blockchains," 2018.



- [13] M. Toulouse, H. K. Dai, and T. G. Le, "Distributed load-balancing for account-based sharded blockchains," *International journal of Web information systems*, vol. 18, no. 2/3, pp. 100–116, 2022.
- [14] Bitquery. (2021, August) Bitquery Builder. [Online]. Available: <https://community.bitquery.io/t/bitquery-builder/>
- [15] R. K. Mondal, E. Nandi, and D. Sarddar, "Load balancing scheduling with shortest load first," *International Journal of Grid and Distributed Computing*, vol. 8, no. 4, pp. 171–178, 2015.
- [16] F. D. Croce and R. Scatamacchia, "Longest processing time rule for identical parallel machines revisited," 2018.
- [17] S. Y. Chang and H.-C. Hwang, "The worst-case analysis of the multifit algorithm for scheduling nonsimultaneous parallel machines," *Discrete Applied Mathematics*, vol. 92, no. 2, pp. 135–147, 1999. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0166218X99000499>
- [18] A. Mizrahi and O. Rottenstreich, "Blockchain state sharding with space-aware representations," *IEEE Transactions on Network and Service Management*, vol. 18, pp. 1571–1583, 2021. [Online]. Available: <https://api.semanticscholar.org/CorpusID:235406552>
- [19] S. Woo, J. Song, S. Kim, Y. Kim, and S. Park, "Garet: improving throughput using gas consumption-aware relocation in ethereum sharding environments," *Cluster Computing*, vol. 23, no. 3, p. 2235–2247, sep 2020. [Online]. Available: <https://doi.org/10.1007/s10586-020-03087-1>
- [20] S. Kim, J. Song, S. Woo, K. Youngjae, and S. Park, "Gas consumption-aware dynamic load balancing in ethereum sharding environments," 06 2019, pp. 188–193.
- [21] M. Li, W. Wang, and J. Zhang, "Lb-chain: Load-balanced and low-latency blockchain sharding via account migration," *IEEE Trans. Parallel Distributed Syst.*, vol. 34, no. 10, pp. 2797–2810, 2023. [Online]. Available: <https://doi.org/10.1109/TPDS.2023.3238343>
- [22] N. Okanami, R. Nakamura, and T. Nishide, "Load balancing for sharded blockchains," Cryptology ePrint Archive, Paper 2020/1466, 2020, <https://eprint.iacr.org/2020/1466>. [Online]. Available: <https://eprint.iacr.org/2020/1466>
- [23] M. Król, O. Ascigil, S. Rene, A. Sonnino, M. Al-Bassam, and E. Rivière, "Shard scheduler: object placement and migration in sharded account-based blockchains," 2021.
- [24] Y. Zhang, S. Pan, and J. Yu, "Txallo: Dynamic transaction allocation in sharded blockchain systems," 2022.
- [25] G. V. Cybenko, "Dynamic load balancing for distributed memory multiprocessors," *J. Parallel Distributed Comput.*, vol. 7, pp. 279–301, 1989. [Online]. Available: <https://api.semanticscholar.org/CorpusID:1680222>
- [26] P. Berenbrink, T. Friedetzky, and R. Martin, "Dynamic diffusion load balancing," in *Automata, Languages and Programming*, L. Caires, G. F. Italiano, L. Monteiro, C. Palamidessi, and M. Yung, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 1386–1398.
- [27] L. Xiao, S. Boyd, and S. Lall, "Distributed average consensus with time-varying metropolis weights," *Automatica*, 01 2006.
- [28] L. Xiao, S. Boyd, and S.-J. Kim, "Distributed average consensus with least-mean-square deviation," *J. Parallel Distrib. Comput.*, vol. 67, no. 1, p. 33–46, jan 2007. [Online]. Available: <https://doi.org/10.1016/j.jpdc.2006.08.010>
- [29] E. Jeannot and F. Vernier, "A practical approach of diffusion load balancing algorithms," in *Euro-Par 2006 Parallel Processing*, W. E. Nagel, W. V. Walter, and W. Lehner, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 211–221.
- [30] C. G. Lopes and A. H. Sayed, "Diffusion least-mean squares over adaptive networks: Formulation and performance analysis," *IEEE Transactions on Signal Processing*, vol. 56, no. 7, pp. 3122–3136, 2008.
- [31] G. Shi and K. H. Johansson, "Convergence of distributed averaging and maximizing algorithms : Part i: Time-dependent graphs," in *2013 American Control Conference (ACC) :*, ser. Proceedings of the American Control Conference. American Automatic Control Council, 2013, pp. 6096–6101, qC 20131104.
- [32] V. Schwarz, G. Hannak, and G. Matz, "On the convergence of average consensus with generalized metropolis-hasting weights," in *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2014, pp. 5442–5446.
- [33] M. Toulouse, H. Dai, and Q. Nguyen, *A Consensus-Based Load-Balancing Algorithm for Sharded Blockchains*, 11 2021, pp. 239–259.
- [34] Etherscan. (2024) Etherscan Documentation. Etherscan. [Online]. Available: <https://docs.etherscan.io/>
- [35] Kaggle. (2024) Ethereum Blockchain Dataset. [Online]. Available: <https://www.kaggle.com/datasets/bigquery/ethereum-blockchain>

