Journal of Universal Computer Science, vol. 18, no. 20 (2012), 2771-2797
submitted: 11/1/11, accepted: 30/4/12, appeared: 1/12/12

Distributed Load Balancing Algorithms for Heterogeneous Players in Asynchronous Networks

Luiz F. Bittencourt, Flávio K. Miyazawa, André L. Vignatti
(Institute of Computing, University of Campinas
Av. Albert Einstein, 1251, Campinas, Brazil, 13083852
{bit,fkm,vignatti}@ic.unicamp.br)

Abstract: In highly scalable networks, such as grid and cloud computing environments and the Internet itself, the implementation of centralized policies is not feasible. Thus, nodes in such networks act according to their interests. One problem with these networks is *load balancing.* This paper considers load balancing in networks with *heterogeneous* nodes, that is, nodes with different processing power, and *asynchronous* actions, where there is no centralized clock and thus one or more nodes can perform their actions simultaneously. We show that if the nodes want to balance the load without complying with certain rules, then load balancing is never achieved. Thus, it is necessary to implement some rules that need to be distributed (i.e., so that they run locally on each node) due to the unfeasibility of centralized implementation. Due to the game-theoretic nature of the nodes, the concept of solution is when all nodes are satisfied with the load assigned to them, a *Nash equilibrium* state. Moreover, we discuss how the rules can be created and present three sets of rules for the nodes to reach a Nash equilibrium. For each set of rules, we prove its correctness and, through simulations, evaluate the number of steps needed to reach the network's Nash equilibrium.
Key Words: load balancing, Nash equilibrium, game theory
Category: F.2.0, I.2.11

1 Introduction

Distributed computing is widely used to process different kinds of jobs in different scientific fields. The development of large, heterogeneous distributed systems such as grids [Foster et al., 2001] and clouds [Hayes, 2008] has brought new challenges, including larger distributed systems and new variables to be tackled when allocating jobs. To achieve high performance in such systems, job distribution across resources (or nodes) is an important issue.

Distributed systems are powerful computing infrastructures that can be used to solve computationally demanding problems. They are geographically distributed over different autonomous users and organizations, which make them potentially large. As a central entity may not be able to handle all these resources, a decentralized resource management is desired. In addition, system resources that are owned by distinct users and organizations may exist for other purposes, which can introduce conflicting interests when making them available to the system. For example, a user may be concerned only with his jobs and resources, and may thus want to get the best resources for executing his jobs and minimize the use of his own resources. Such behavior can be seen as a game where each user is interested in running his jobs faster using other users' resources.

The large scale of distributed systems in conjunction with the users' game theoretic behavior motivated the development of the load balancing algorithms presented in this paper. Load balancing algorithms try to ensure that every node in the system has the same amount of work. To achieve this, the algorithm assigns or moves jobs (or job parts) between nodes, thus diminishing the amount of work in one node to augment it in another one. A finite sequence of job movements is expected to achieve fair sharing of jobs over the nodes, with each node having an amount of work compatible with its processing capacity.

More specifically, this paper considers the problem of balancing the load on the nodes of a network. The model is motivated by the unfeasibility of designing centralized load balancing algorithms. This unfeasibility can be caused by several reasons, such as high system scalability and heterogeneity. Due to this lack of centralized control, our model assumes that each node is an independent entity with its own goal. In addition, if a centralized control attempts to implement any kind of rule, the entity can simply refuse to enter the network if these rules are too complex or counter intuitive. Even if entities agree to obey certain rules, the network's high scalability renders the design of a centralized algorithm unfeasible in practical situations. Therefore, throughout the text, we define rules that are *distributed* and meet the goal of each entity.

From a global point of view, the system aims to balance its load. The users, however, want to remain in a state where they are satisfied with the load assigned to them, otherwise they do not have incentive to enter the network. If all users are satisfied, the load exchange between them stops and they can concentrate on processing of their loads. Thus, due to the lack of central control, user satisfaction is more important than load balancing, which motivates the use of a game theoretic approach. Therefore, we say that load balancing converges to a solution where all users are satisfied, which we call a *Nash equilibrium* state. We therefore present some sets of rules to be used by users such that each set of rules takes the users to a Nash equilibrium state. As a comparison metric for each set of rules we present, we use the number of steps required in simulations to reach a Nash equilibrium.

A key concept in this work is the lack of central control, which is the main motivation for using Nash equilibrium as the solution concept. If there is no central control, the system becomes anarchic. In an anarchic system, the participating agents cannot make predictions about the system's behavior. Therefore, it is expected that these agents accept to follow certain rules to obtain good operation of the system. From one point of view, they are cooperating. But not all types of cooperation are accepted in the system: The simpler the rule, the better the acceptance. I.e., in face of complex rules, the nodes do not cooperate: They simple choose to not enter the network. In this paper, the nodes want a low load. Without rules, however, the nodes cannot estimate the fluctuations of the future load: A node can thus suddenly receive a very high load. In addition, exchanging loads too frequently can delay the execution of all jobs, which is not desirable for any node in the system. Thus, the nodes agree that equilibrium is desirable. They do not fight against equilibrium, but not all the designed rules are well accepted.

There are several motivations for the nodes not accepting some rules: Good rules are not accepted because they are intuitively harmful (e.g., transferring low load is intuitively worse than transferring high load), they are difficult to follow (e.g., not all algorithms are easily designed for asynchronous systems), or they are impossible to follow (e.g., the set of web services available to the agents does not include a procedure to search for information about the neighbors of neighbors). Therefore, from this point of view, the nodes resist to cooperate, that is, they won't accept any rule (a central authority could mandate that). In addition, if a node does not accept to balance the load, it is bad for the node itself, because a computation that concerns this node (e.g., triggered by the node owner) may never be completed if nodes just want to get rid of their load (without performing any computation). In this sense, a well designed load balancing rule can speed up the computation of interest to that node. A more practical example is the case of file sharing applications, such as BitTorrent [Golle et al., 2001, Feldman et al., 2004, Cohen, 2003] and eMule [Heckmann et al., 2005]. In these systems, all users want to download files without uploading files. Thus, over the years, many rules were added to file sharing protocols so that the users not only download but also upload files. There are many scientific papers that study which rules need to be added to obtain a "fairer" file sharing system without scaring users away.

To illustrate this, consider the following example, based on a BitTorrent network penalty scheme. Suppose we have a wireless network to which users can connect using their laptops. However, users must somehow "pay" to access the Internet using this wireless network. Assume that "processing load" is a valid currency to pay for Internet access in this network. Users can accept jobs or simply forward all jobs to their neighbors. In this scenario, if the wireless network owner wants to run jobs and if all users help, each user processes a small part and then can access the Internet. The network owner makes each task store where and for how long it was executed. This information tells the network owner how much each laptop collaborated in comparison to the average. If the collaboration of a certain node is below average, considering a standard deviation, the owner can slow down the connection speed for that node as a penalty. Therefore, each node wants to process the load balancing average to avoid unnecessary effort processing more than the average. On the other hand, if a user refuses too much load (i.e., forwarding many jobs to neighbors) the owner will know and will apply a penalty (as in the BitTorrent network), and the user's Internet access would incur prejudice. In such a scenario, each laptop would try to process exactly the average.

1.1 Related Work

In their essence, load balancing algorithms can be centralized or decentralized, and static or dynamic. A centralized load balancing algorithm has a central entity that concentrates information about the system nodes and is responsible for assigning jobs (load) to them [Lin and Raghavendra, 1992]. On the other hand, a decentralized algorithm [Riska et al., 2000] runs on all nodes, and each node has a partial view of the system and thus makes a local decision to achieve load balancing. A static algorithm [Kim and Kameda, 1992] uses unmodified information gathered before its execution to achieve load balancing, while a dynamic algorithm [Lu et al., 2006] can use information updated during load balancing and/or the execution of jobs to make its decisions. Hybrid (or semi-static) approaches exist as well [Subrata et al., 2008]. This paper proposes algorithms for decentralized dynamic load balancing.

This work does not consider dependent tasks (such as those represented by workflows or directed acyclic graphs) during load balancing. However, a meta-scheduler such as Condor DAGMan [CondorTeam, 2010] could be used to select, at a higher level, tasks that are ready to be executed independently (i.e., tasks that have all their precedences already satisfied, with no dependencies between them). Dependent tasks could thus be handled by a DAG/workflow manager while the load balancing algorithm deals only with independent tasks.

Many papers deal with load balancing in a game-theoretical framework. For non-distributed systems, Even-Dar et al. [Even-Dar et al., 2007] study convergence time to reach a Nash equilibrium in load balancing problems in an *elementary step system* (ESS), and show lower and upper bound results for many cases. Goldberg [Goldberg, 2004] considers a weakly distributed protocol that simulates the ESS. In the author's protocol, a task chooses machines at random and migrates if the load in the chosen machine is lower. He uses a potential function to show upper bounds in the number of steps to reach Nash equilibrium. Even-Dar and Mansour [Even-Dar and Mansour, 2005] consider the case where all users choose to migrate at the same time in a real concurrent model. In this model, tasks migrate from overloaded to underloaded machines according to probabilities computed by considering that the load information of all machines is known. Berenbrink et al. [Berenbrink et al., 2006] propose a strong distributed protocol that needs very little global information, where a task needs to query the load of only one other machine, migrating if that machine has a smaller load.

All these works consider a complete network model where a node can send load to any other network node; this work, however, considers a more general model with arbitrary network topologies. As far as we know, this is the first work that considers a non-cooperative game-theoretical load balancing model with arbitrary network topologies. Another difference between this work and others is that in our model the resources, instead of the jobs (loads), are the players. This fact better represents real-world distributed systems, because a job owner assigns tasks to any resource that can execute them, without worrying about overloading the resource. In other words, resource overload is a problem to be addressed by the resource itself, and not by the job owner.

1.2 Contributions

This work's main contributions are the definition of a network model that is a closer representation of real-world networks and three sets of rules the nodes must follow to reach Nash equilibrium. We prove the correctness of each set of rules presented. The sets of rules are evaluated through simulations that compare the number of steps required for the nodes to reach a Nash equilibrium. In addition, we discuss how node rules can be created.

In our model, each entity is represented by a node, and a node has the goal of minimizing its load by sending part of it to its neighbors in the network. We assume that a node is *satisfied* if its load is (approximately) equal to the minimum load of its neighbors. The nodes will never be satisfied if this weak assumption is not considered (more details about this are presented in Section 2). We aim to achieve a state where all nodes are satisfied, thus the load exchanges stop, either arriving at or leaving the node, and the load can be processed without interruption. If all nodes are satisfied, we are in a *Nash equilibrium* [Nash, 1951] state. For technical reasons, which we discuss in Section 2, reaching an exact Nash equilibrium is not always feasible, and therefore we consider an approximation to the equilibrium.

We consider that the load can be divided into infinitesimal parts. Hence, if the system is in approximate Nash equilibrium, then the loads of the nodes are approximately equal to those in optimal load balancing. Therefore, questions about the *price of anarchy* [Papadimitriou, 2001] and similar measures are not interesting in this model. Moreover, by setting the load of the nodes equal to those of optimal load balancing, the system remains in equilibrium because no node has incentive to send load to other nodes. Therefore, computing the loads of a Nash equilibrium is another trivial task. Real-world applications that can be arbitrarily split into smaller parts are the main focus of our algorithms. For example, image processing filters can usually be applied to arbitrarily small parts of an image and then merged to compose the resulting image. Such applications are common in physics, chemistry, computer science, and biology.

As stated before, the rules must comply with the goal of the nodes and be implemented in a distributed way, that is, implemented at each node. Complex rules and/or rules that do not explicitly comply with the goal of a node can be simply ignored by the node. Thus, this work seeks to create rules that are simple enough that the nodes agree to follow them.

In *heterogeneous* systems, nodes have different kinds of hardware and software. On the other hand, in *homogeneous* systems, all nodes are considered to have the same computing capacities. The model described in this work considers different processing speeds at each node to represent a heterogeneous system. In highly scalable networks, the assumption of *synchronous* actions is unrealistic. In this case, the node actions can be performed sequentially or simultaneously. In practice, however, we cannot assume that all users act simultaneously or sequentially. What really happens is a mix between the ESS [Even-Dar et al., 2007], where at a given time instant only one node performs an action, and the parallel case, where all actions are performed simultaneously [Berenbrink et al., 2006, Even-Dar and Mansour, 2005]. The situation is more problematic when the actions occur simultaneously, because the subsequent state of the system is harder to predict than when only one player decides at a time instant. In the *asynchronous* case, some of the actions occur simultaneously, which, besides being a more realistic scenario, is a more general case than the ESS or parallel case. This work demonstrates that the rules proposed are also valid for the asynchronous case.

Some kinds of games assume that users communicate false information, that is, they lie. We do not consider this case here, since it falls under the subject of auctions. However, in our model, a way to avoid lies is if each node "estimates" a neighbor's load instead of directly asking it. This approach can be implemented, for example, by sending a small load to see how the neighbor reacts to it. Another way is to observe the load the neighbor sends, given that all the nodes follow the same algorithm. In addition, considering the last example given in Section 1, the tasks themselves communicate to the network owner which node executes a given task, avoiding lies.

1.3 Paper Structure

The paper is organized as follows. Section 2 defines the system model and the problem, as well as explains why players need to follow rules for the system to converge to a Nash equilibrium. Section 3 discusses how the rules (algorithms) can be designed to meet the players' objectives. Section 4 presents three distributed algorithms and proves that they reach a Nash equilibrium. Section 5 shows the results of an empirical comparison between the three proposed algorithms. The conclusion and avenues for future work are presented in Section 6.

2 The Model

The model is composed of a network represented by a connected graph $G = (V, E)$, where each node $i \in V$ has a *processing weight* $p_i > 0$ and a *processing speed* $s_i > 0$. Let $\ell_i = p_i/s_i$ be the *load* of node i. If two nodes are connected by an edge, we say that they are *neighbors* of each other, and they can send an unlimited amount of processing weight through that edge. Throughout this study, when a node sends processing weight to another node, by abuse of language, we simply say that this node *sends load* to another node. Each network node is controlled by a player who wants to minimize his load as well as complete his jobs. To do this, the player can transfer an amount of load to his neighbors. In practical situations, it is not feasible for a node to know the information of all the other nodes in the network. Therefore, our model assumes that a node (player) has only the load information of its neighbors and the load of the other nodes is unknown.

We refer to a network or system as *homogeneous* when all s_i are equal; otherwise we refer to it as *heterogeneous*. In homogeneous systems, we consider without loss of generality that $s_i = 1$, and therefore $\ell_i = p_i$.

The players could simply send all of their loads to their neighbors, but this would not be a good strategy, as we show in the following example. Assume a homogeneous network with two nodes i and j connected by an edge, with $\ell_i = 200$ and $\ell_j = 100$. Thus, if both players simultaneously send all their load to their neighbors, we have $\ell_i = 100$ and $\ell_j = 200$; if only one node sends its load, the other node load will be equal to 300. Therefore, regardless whether or not the players perform their actions simultaneously, unstable load exchange occurs indefinitely, which leads to an unpredictable delay in the execution of the players' own jobs. Thus, the players are willing to send their loads to available neighbors, and do so with a minimal number of messages. Therefore, each player needs to follow some *rules*.

The first rule says that a player is *satisfied* (i.e., has no incentive to send load to his neighbors) if his load in the corresponding node is equal to the minimum load of its neighbors. Let $N(i)$ be the set of neighbors of node i. In other words, we say a player i is satisfied if $\ell_i \leq \ell_j$, $\forall j \in N(i)$. We say that the system is in *Nash equilibrium* if all players are satisfied in that state.

In practice, the continuous divisions of the load can cause infinitesimal transfers between nodes, and an exact Nash equilibrium cannot be reached in a finite number of steps. Thus, we use a relaxed definition of the equilibrium, called *ε-Nash equilibrium*, that is best suited to our model.

Definition 1 Fundamental rule. A player i is *ε-satisfied* if $\ell_i \leq (1 + \varepsilon)\ell_j$ for all $j \in N(i)$.

Definition 1 is called the *fundamental rule* because without it a player would only be satisfied with a load equal to zero and, consequently, the system would always be unstable, without performing any useful computations. Using the above definition, we can define the ε*-Nash equilibrium.*

Definition 2 ***ε-Nash equilibrium.*** The network is in ε*-Nash equilibrium* if all players are ε-satisfied.

The ε value is chosen according to which values the nodes consider to be small, that is, the load amount that does not significantly affect a node's processing time. For example, if 1 KB is considered to be small and the nodes have a load of at most 1,000,000 KB $=$ 1 GB, then, in our model, considering that the load in each node does not exceed 1,000, we can set $\varepsilon = 0.001$. Section 5 uses these values in the simulations.

What really interests us, and the reason we established the fundamental rule, is defining the rules that will be built into the system so that, given an arbitrary initial allocation of loads on the nodes, the nodes act in a way that the system reaches a Nash equilibrium. Using only the fundamental rule is not sufficient to achieve system equilibrium. For example, using the same two-node example presented above, we fall back on the erratic behavior of load exchanges, independent of the load amount sent by the player until he becomes satisfied and whether or not the actions are performed simultaneously. Thus, we need to create other rules for the system to reach and remain in a Nash equilibrium.

3 Designing the System Rules

As argued in Section 2, the fundamental rule of Definition 1 is not a sufficient rule to ensure that players reach a Nash equilibrium. In this case, we must create new rules for the players, and these rules must be simple and distributed. To guide the creation of these rules, we focus on the answer to three key questions:

- **Who** receives load?
- **How much** load should be sent?
- **How** should load be sent?

The first question concerns which nodes to select to send the load. As stated in Section 2, we know that a node can only obtain information and send load to its neighbors. We would therefore like to know which neighbors can be selected to receive load. The second question tries to define the load amount that is removed from the node in question to send to the selected neighbors. This amount depends on the extent to which the node is overloaded with respect to its neighbors. Given that we have a candidate set to receive a load amount available to be sent, we can answer the third question in several ways. For example, we can divide the load between the neighbors that will receive load or we can simply send it to a single node.

To better focus on the ideas, throughout this section we deal with *homogeneous* networks, that is, when $\ell_i = p_i$. Section 4 concretizes the ideas of this section and then considers the heterogeneous case.

3.1 Who receives load?

A node i must select a subset of nodes $N(i)$ to send load. To perform this selection, i is restricted to only the information of nodes in $N(i)$. Since the goal is to balance the load, obviously heavier nodes should send load to lighter nodes. Since the rules (algorithms) must be the same for all nodes, from the local point of view of a node i, if the rule does not select neighbors j such that $\ell_j < \ell_i$ to send load to, the system load will not get balanced. Therefore, the rules must necessarily consider load transfer to lighter nodes.

A counter intuitive approach is the case when node i selects a neighbor j to send load to when $\ell_j \geq \ell_i$. This is a valid approach if j can receive load if we visualize the network globally. For instance, consider a network with the path topology formed by three nodes a, b, and c, where a is connected with b and b is connected with c. A Nash equilibrium is obtained when all the loads are equal to $\nu = (\ell_a + \ell_b + \ell_c)/3$ (for the purposes of the example, we consider an exact equilibrium). If $\ell_a > \nu$, $\ell_b > \ell_a$, and $\ell_c < \nu$, then we know that some load in a must go to c. In this case, it is valid to send load from a to b, even with $\ell_b > \ell_a$. On the other hand, if $\ell_a < \nu$, $\ell_b > \ell_a$, and $\ell_c < \nu$, then we know that if a sends load to b, at some point that same load amount will return to a, making the convergence to equilibrium slower. In other words, a node's local view makes it impossible to determine whether or not it is a good strategy to send load to a more loaded node.

We can further improve the use of neighborhood information to estimate the load of the equilibrium. The idea is to calculate the value of a "local equilibrium" and use it to select nodes. For this, a node i calculates the local mean of the loads, which is given by $\overline{x} = (\ell_i + \sum_{j \in N(i)} \ell_j)/(|N(i) + 1|)$. Thus, i selects only nodes j such that $\ell_j < \overline{x}$ to send the load to. This strategy is justified by the fact that if we send load to nodes with load above the "local equilibrium", then much of that load may return to the node later. Therefore, we use a larger amount of local information to achieve faster global load balancing, but the load may or may not return to the node, depending entirely on the network configuration.

Summarizing the above discussion, we must always design rules that select lighter nodes to send load to, but we can also select heavier nodes. Additionally, the selection can use information from the local mean in an attempt to send less load than necessary, since the load may later have to return to the node and thus slow down the convergence. Moreover, we can try specific combinations of these strategies, for example, sending load only to lighter nodes that are below the local mean.

In Section 4.1, the proposed method uses information from all neighbors to calculate a local mean and to decide which neighbors will receive load. In Sections 4.2 and 4.3, the load is sent to the lighter nodes, without using the mean information.

3.2 How much load should be sent?

Since the nodes only have neighborhood information, we are never sure about the ideal *amount* to be sent to neighbors. As a rule, when designing algorithms that send small amounts of load, the convergence can be very slow but with small load fluctuations. On the other hand, if large amounts of load are sent, the convergence may be faster but the system may be less stable, in the sense that the system does not even reach a Nash equilibrium or has high load fluctuations, which, in turn, can have a negative effect on convergence speed. To illustrate this, we present an example in Figure 1.

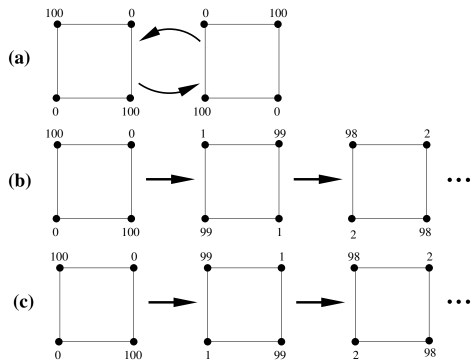


Figure 1: Example

In Figure 1, we consider simultaneous actions. Case (a) occurs when the nodes are extremely aggressive and in every step send all their loads to their neighbors; in this case, a Nash equilibrium is not reached. Case (b) occurs when each node sends almost all of its load ($\approx 99\%$) to its neighbors. In this case, there are high load fluctuations; however, a Nash equilibrium is slowly reached. In case (c), the nodes act conservatively by sending low amounts of load ($\approx 1\%$); in this case load fluctuations do not occur and a Nash equilibrium is slowly reached. If we send a medium amount of load, we avoid load fluctuations and achieve faster convergence. However, a high amount of load sent on one network can prove to be very bad for another (see Figure 2).

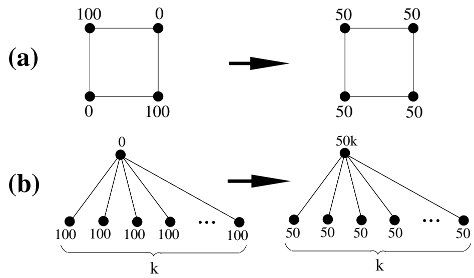


Figure 2: Example

The initial network, presented in Figure 2(a), is identical to the initial network of Figure 1, but in this case, the nodes send 50% of the load to neighbors in an attempt to speed up the convergence and avoid high fluctuations. As Figure 2(a) shows, after one step, a Nash equilibrium is obtained. However, if 50% of the load in each node is sent to neighbors in the network of Figure 2(b), the result is very poor, because the node that originally had a load equal to 0 will now have a load equal to $50k$. Since $k = O(n)$, when we use the same strategy, we can get a balance that is $O(n)$ times worse than that achieved in the previous step.

In the examples presented, it is clear that we have a tradeoff between the amount of load sent and the convergence time to a Nash equilibrium. Moreover, the high load fluctuations (Figure 1(b)) and the accumulation of load at the nodes (Figure 2(b)) pose serious problems in convergence speed and make it difficult to determine the amount of load a node needs to send.

3.3 How should load be sent?

If the subset of neighbors that will receive load and the amount of load to be sent are already defined, the next step is to define how the load will be sent. Let i be a player, S the subset of neighbors of i that were selected to receive load, and Δ the amount of load that i will send. An example of a strategy is to send $\Delta/|S|$ to each node in S. This strategy has the advantage of being simple and “fair”, since all nodes in S receive the same amount of load and no nodes are privileged. On the other hand, since we seek a rapid convergence to a Nash equilibrium, we can send Δ only to the node whose load is minimal (possibly more than one node).

In this case, the least loaded nodes are quickly filled, therefore speeding up the convergence. Furthermore, a low-loaded node may receive much more load than necessary because it has become a “target” for its neighbors to send their loads to. This can lead to problems of load accumulation, as seen in the example of Figure 2(b).

Both of the discussed strategies have problems. In the egalitarian division strategy, lighter nodes have no priority in receiving load, and therefore the convergence may be slow. In the “best response” strategy, the lighter nodes have priority in receiving load, causing load to accumulate at these nodes, possibly unbalancing the network even further. In an attempt to bypass these problems, we describe the *water-filling* strategy, a mix of the two previous strategies.

```
input: Δ, S.
begin
    Sort S = {i_1, ..., i_m} such that ℓ_{i_1} ≤ ℓ_{i_2} ≤ ... ≤ ℓ_{i_m}
    L = 0,  k = 0,  ℓ_{i_{m+1}} = ∞,  ts = 0
    while Δ > 0 do
        k = k + 1
        ts = ts + s_{i_k}
        h = min{Δ/ts , ℓ_{i_{k+1}} − ℓ_{i_k}}
        L = L + h
        Δ = Δ − h · ts
    for j = 1 to k do
        send s_{i_j}(L − ℓ_{i_j}) to node j
end
```

Algorithm 1: Water filling

Algorithm 1 describes the water-filling technique, executed by each node i, which receives Δ and S as input. The idea behind this technique is to equally distribute the load at the lightest nodes in S. At the end of each step of the while loop, the load of the k nodes with lowest loads are increased until the load of the $k+1$-th node, or all the load in Δ is totally distributed. The name of this technique is due to the fact that it resembles the situation in which a liquid is poured to fill a container and the lowest levels of the container get equally filled. Figure 3 shows an example of Algorithm 1's execution. Initially, in Figure 3(a), we have $\Delta = 100$; after executing the water-filling algorithm, we have the configuration in (b).

Note that Algorithm 1, can be implemented to execute in $O(N \log N)$ time (due to the sorting step) and $O(N)$ message complexity.

In short, there are three options for sending load. *Egalitarian division* is simple and “fair”; however, by not setting priorities to nodes, it may have a slow convergence rate. This strategy is used in Section 4.1. The *best response* method gives preference to the least loaded nodes but can lead to load accumulation. The

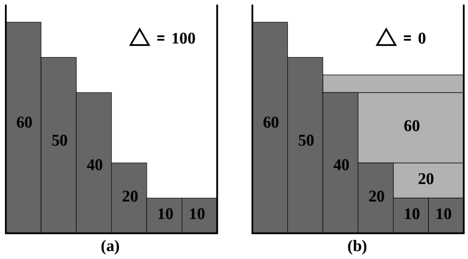


Figure 3: Example

best response strategy is used in Section 4.2. The *water-filling* technique is a mix of the other two strategies, with the load equally and progressively distributed between the least loaded nodes; this strategy is used in Section 4.3.

4 Distributed Algorithms for Load Balancing

Section 3 discusses how to design simple and distributed rules that can be used by nodes to reach a Nash equilibrium. The discussion focused on three key questions and presented some ideas and strategies. Much of the discussion purposely did not formalize or concretize these ideas, because we believe that simply presenting the algorithms can avoid the rich discussion of Section 3 about the many possibilities on how to design system rules. Furthermore, the contents of Section 3 explain"how to think" when designing such rules, which can help and guide in future works on this subject.

This section aims to concretize the discussion of Section 3 by presenting three *distributed* and *asynchronous* methods to be used by the nodes in *heterogeneous* systems. In each method, we present proof of convergence to a Nash equilibrium. Later, in Section 5, we perform empirical tests to compare the convergence speeds of these methods.

Some details about the asynchronous environment are worth discussing. For example, to obtain neighbors' load information, a user sends parallel requests to them for the information. A reliable protocol (i.e., one that guarantees message delivery) is used in the communication between nodes, and parallel requests are repeated if any of the responses are not received before a timeout.

Due to the system's asynchrony, the responses arrive with different delays. Thus, if a neighbor updates its load, its response may contain obsolete information. In this case, a node needs to synchronize its neighbors' responses. However, such an implementation is complicated and, therefore, it is a deciding factor for users in sticking with the proposed protocols. Therefore, it is possible to receive obsolete information, but it only interferes with the algorithm's correctness if it occurs very often. Experiments show that the presented algorithms converge to a solution even if the information used in the computation of neighbors' loads is obsolete.

Some proofs need a chronological time notion. A *time instant* is the point in time when one or more actions are performed simultaneously. Given a time instant t, the time instant $t+1$ is the time instant following t.

In each of the following methods, the algorithm presented is used by a given node i. We use S to denote the subset of neighbors to which i sends its load, Δ to denote the amount of load i will send, and $\overline{x}$ to denote the mean over a specific subset of neighbors. Each method redefines S, Δ, and $\overline{x}$. In addition, by simplicity, we consider an ε-*satisfied* player i, one who satisfies $l_i \leq (1+\varepsilon)^3 l_j$, since we can use $\varepsilon' = \varepsilon/7$ instead of ε. In this case, an ε'-*satisfied* player with this condition is also an ε-*satisfied* player with the original condition. We note that there is no other intuition in redefining the ϵ-satisfied player; this is done only to directly fit in Definitions 5 and 6 of the bounded step method of Section 4.2, which are based on the definitions used in [Awerbuch et al., 2008].

4.1 Egalitarian Method

This section describes the *egalitarian method*, presented in Algorithm 2. The design of Algorithm 2 is motivated by simplicity and the use of all available information to calculate a "local equilibrium", as explained in more detail in Section 3.1. The reason to use the "local equilibrium" to define Δ is to prevent the load from returning to the original node. Thus, if we use a more "aggressive" division of the load rather than an egalitarian one, it could cause load accumulation at the nodes, and hence the goal of preventing the load from returning to its origin would not be achieved. In short, the egalitarian method distributes load in a conservative way, trying to prevent the load from returning and possibly speeding up the convergence process.

In the egalitarian method, the set S is composed of only nodes that make i unsatisfied, that is, $S = \{j \in N(i) : (1+\varepsilon)^3 \ell_j < \ell_i\}$.

Since we consider the heterogeneous case, $\overline{x}$ is a local mean given by the sum of the processing weights p_j divided by the total processing speeds s_j, but since we consider an ε-approximation of the equilibrium, the definition of $\overline{x}$ needs to express this approximation as well. Let $S_= = \{j \in N(i) : \ell_i/(1+\varepsilon)^3 \leq \ell_j \leq (1+\varepsilon)^3 \ell_i\}$ and $S_{\neq} = N(i) \setminus S_=$. Therefore, $\overline{x}$ is given by $\overline{x} = \frac{p_i + \sum_{j \in S_=} p_i + \sum_{j \in S_{\neq}} p_j}{s_i + \sum_{j \in S_=} s_i + \sum_{j \in S_{\neq}} s_j}$. In other words, the nodes in $S_=$ are considered to have their load equal to ℓ_i, since they can neither send nor receive load from i (due to the definition of S).

Moreover, Δ is defined as the "excess" of the processing weights of node i regarding the mean, that is, $(\ell_i - \overline{x})s_i$. The method is called egalitarian because Δ is proportionally divided (considering the speeds s_j) between the nodes in S

such that all the neighbors receive the same amount of load.

```
begin
    Let S_= = {j ∈ N(i) : ℓ_i/(1+ε)^3 ≤ ℓ_j ≤ (1+ε)^3 ℓ_i}
    Let S_≠ = N(i) \ S_=
    Let x̄ = (p_i + Σ_{j∈S_=} p_i + Σ_{j∈S_≠} p_j) / (s_i + Σ_{j∈S_=} s_i + Σ_{j∈S_≠} s_j)
    Let Δ = (ℓ_i − x̄) s_i
    if Δ > 0 then
        Let S = {j ∈ N(i) : (1+ε)^3 ℓ_j < ℓ_i}
        for j ∈ S do
            Send Δ s_j / (Σ_{k∈S} s_k) from i to j   /* decreases Δ */
end
```

Algorithm 2: Algorithm for player i

In Algorithm 2, note that each node j receives the same amount $\frac{\frac{\Delta s_j}{\sum_{k\in S} s_k}}{s_j} = \Delta/(\sum_{k\in S} s_k)$ of load; this justifies the name *egalitarian.*

As an example, consider two nodes in a heterogeneous network, where $p_i = 200, s_i = 2, p_j = 100, s_j = 3$, and ε is very small. Thus, $\overline{x} = (200+100)/(2+3) = 60$ and $\Delta = (100-60)\cdot 2 = 80$. Therefore, if i sends 80 to j, the system reaches a Nash equilibrium.

We must show that Algorithm 2 reaches equilibrium. A technique often used to show the convergence to equilibrium is the potential function that sums the squares of the loads [Even-Dar et al., 2007]. However, this technique cannot be used in our case, as shown in the following example. Consider a homogeneous network with a star-like topology with nodes a, b, c, d, and e and edges $(a, b), (a, c), (a, d)$, and (a, e). Initially $\ell_a = 0$ and $\ell_b = \ell_c = \ell_d = \ell_e = 10$. Thus, the sum of the squares of the loads is $0+100+100+100+100 = 400$. We assume that all nodes act simultaneously; thus after one step of Algorithm 2, we have $\ell_a = 20$ and $\ell_b = \ell_c = \ell_d = \ell_e = 5$, and the sum of the squares of the loads is equal to $400 + 25 + 25 + 25 + 25 = 500$. Using the same idea, after another step, a Nash equilibrium is reached and the sum of the squares of the loads is equal to 320. In other words, this function can increase or decrease after a step, making it unsuitable for measuring the progress to a Nash equilibrium.

Lemma 3. *Let ℓ_i^t be the load of node i at time t. Let $L_t = (\ell_{v_1}^t, \ell_{v_2}^t, \ldots, \ell_{v_n}^t)$ be the array of the loads at time t sorted in non-decreasing order. If at time t there is at least one node with $\Delta > 0$, then L_{t+1} is lexicographically greater than L_t.*

Proof. The nodes that send load can have their load decreased at time $t + 1$; therefore these are the nodes that hinder the demonstration. This proof considers the case where all ℓ_i are distinct but, using the same idea, a similar proof can be carried out for the case where not all ℓ_i are distinct.

Let Q be the set of nodes that send load (i.e., those nodes with $\Delta > 0$) in time t. By hypothesis, $Q \neq \emptyset$. Note that a node in Q can also receive load at time t. Let $q = \text{argmin}_i\{\ell_i^{t+1} : i \in Q\}$. That is, among the nodes that sent load at time t, q is the node that ends up with the lowest load at time $t+1$.

Suppose that q is in the k-th position of the array L_{t+1}. Let $A_t = (\ell^t_{v_1}, \ell^t_{v_2}, \ldots, \ell^t_{v_{k-1}})$ be the subvector of the $k-1$ first positions of L_t. We say that a node i *belongs* to A_t if its load is among the loads of the array A_t. In the way that q is chosen, the nodes belonging to A_{t+1} only receive load (without sending) or remained unchanged at time t. Thus, all the values in A_{t+1} are greater than or equal to the corresponding values of A_t.

Next, we consider two cases. (i) A node i belongs to A_{t+1} such that i receives load at time t. In this case, i belongs to both A_t and A_{t+1} and its load increases at time $t+1$. Therefore, at least one value in A_{t+1} is strictly greater than one value in A_t, and therefore L_{t+1} is lexicographically greater than L_t.

In the complementary case, we have that (ii) all nodes belonging to A_{t+1} neither receive nor send load at time t. Thus, all values of A_t and A_{t+1} are equal. In this case, we show that the k-th position of L_{t+1} is strictly greater than the same position in L_t. Let s be the less loaded node at time t that receives load from q. Since s receives load, it cannot belong to A_t. Thus, at time t, s will be at least in the k-th position. Therefore, the value of the k-th position of L_t is at most ℓ^t_s. Note that $\ell^{t+1}_q \geq \ell^t_q - \Delta = \overline{x^t_q} > \ell^t_s$, where $\overline{x^t_q}$ is the mean value for node q at time t, and the last inequality is true because s receives load from q at time t. Therefore, the value of the k-th position increases, and therefore L_{t+1} is lexicographically greater than L_t.

Note that this proof is valid for any number of players who send at any given time; therefore it is valid for the asynchronous case. $\square$

Theorem 4 Convergence. *In asynchronous and heterogeneous networks, if the nodes use the egalitarian method described in Algorithm 2, then the system converges to an ε-Nash equilibrium.*

Proof. If the system is not in Nash equilibrium, then at least one node is unsatisfied. Among the unsatisfied nodes, let i be the most loaded. By the choice of i, all $j \in N(i)$ have $\ell_j \leq (1+\varepsilon)^3\ell_i$. Moreover, since i is unsatisfied, then at least one $j \in N(i)$ has $\ell_j(1+\varepsilon)^3 < \ell_i$. When $\overline{x_i}$ is calculated, the nodes $j \in N(i)$ such that $\frac{\ell_i}{(1+\varepsilon)^3} \leq \ell_j \leq (1+\varepsilon)^3\ell_i$ have their load rounded to ℓ_i. Therefore, $\ell_i > \overline{x_i}$ and, consequently, $\Delta = \ell_i - \overline{x_i} > 0$. Thus, we can use the result of Lemma 3, which guarantees that the vector of loads sorted in non-decreasing order in the next time moment is lexicographically greater than the vector of the current step. Since such a vector has a maximum value, we know that the process converges to an ε-Nash equilibrium. $\square$

4.2 Bounded Step Method

The aim behind the method presented in this section is to bound the load amount that a player can send and receive at a given time. Thus the system avoids high fluctuations and load accumulation, hoping to quickly converge to an equilibrium. The method is inspired by the techniques used by Awerbuch et al. [Awerbuch et al., 2008], who obtain a poly-logarithmic upper bound for another load balancing problem. Although these authors assume some rules in a general, they do not consider how these rules can be implemented in practice. We note that although these rules are easy to state, their implementation is more problematic, especially in the case of asynchronous systems. More specifically, we define two rules that users should follow, called the *inertia* and *bounded step* rules [Awerbuch et al., 2008]. After that, we discuss how these rules can be implemented and we present algorithms for their implementation.

Definition 5 Inertia rule. A node i can only send load to a neighbor j if $\ell_i > (1+\varepsilon)^3 \ell_j$.

In other words, the inertia rule ensures that we can only send load between two nodes if their loads differ significantly.

Definition 6 Bounded step rule. Let ℓ_i^t be the load of node i at time t. Then

- $\ell_i^t/(1+\varepsilon) \leq \ell_i^{t+1} \leq (1+\varepsilon)\ell_i^t$, and
- $\ell_i^t \geq \eta$, where η is a small number.

The bounded step rule consists of two constraints. The first ensures that a node's load does not vary too much, and the second one guarantees that we have at least a small load amount at a node during execution. The second is a technical constraint because if $\ell_i^t = 0$, then $\ell_i^{t+k} = 0$ for all integer $k \geq 1$. If the nodes obey these two rules, then they converge to a Nash equilibrium, as shown in Theorem 7.

Theorem 7 Convergence [Awerbuch et al., 2008]. *If the load exchanges are non-null and satisfy the inertia and bounded step rules, the system converges to an ε-Nash equilibrium.*

Next, we explain how the inertia and bounded step rules are implemented at the nodes. We describe two algorithms, called `SEND-RECEIVE` and `LISTENER`.

To not violate the bounded step rule, a node cannot just send an arbitrary load amount to another node. Thus, a sender node needs to communicate how much it is willing to send, and the receiver node can then communicate to the sender how much it can accept. In the `SEND-RECEIVE` algorithm, the `SEND` part deals with "trying" to send the load, and the `RECEIVE` part deals with how much the node can accept. The real sending of load is performed by the `LISTENER` algorithm.

As discussed above, in the `SEND` part of `SEND-RECEIVE`, the load is not immediately sent; what happens is the transmission of an "announcement" from a node i to a node j stating the load amount i is willing to send to j. When i announces the load transfer to j, i moves an amount of load to a *buffer* b_j in order to "reserve" this amount for j. To not violate the bounded step rule, the reserved load b_j is an overestimation of the amount that i is willing to send to j; this is discussed later, in Lemma 9. Since the reserved load is still at i, ℓ_i does not change. The announced load is sent to j only when after j answers the announcement of i, stating the load amount that j will accept from i. In other words, i sends a message to j stating how much load it is willing to send, j sends a reply message stating the fraction of load it is willing to receive, and then i finally sends the load that j is willing to receive. This message exchange is needed to ensure the correct working of the asynchronous protocol.

According to the inertia rule, node i can only send load to a node $j \in N(i)$ when $(1+\varepsilon)^3\ell_j < \ell_i$. In addition, i does not send load to j if the buffer b_j is not empty, since this means that i already tried to send load to j but j has not yet answered. Therefore, the set S of the nodes that can receive load from i is defined as $S = \{j \in N(i) : (1+\varepsilon)^3\ell_j < \ell_i,\ b_j = 0\}$.

To define Δ, we must respect the bounded step rule, which says that $p_i^t - \Delta \geq p_i^t/(1+\varepsilon)$ (actually, the bounded step rule is defined for ℓ_i, but using p_i produces an equivalent definition). Therefore, $\Delta \leq p_i^t(1 - \frac{1}{1+\varepsilon})$, but $\sum_{j\in N(i)} b_j$ is already reserved to be sent to other nodes. Let $d_i = p_i - \sum_{j\in N(i)} b_j$ be the available load amount that i can send to other nodes. Therefore, Δ is defined as $\Delta = d_i(1 - \frac{1}{1+\varepsilon})$.

We also need to determine how Δ is divided among the nodes in S. In this case, Δ is equally divided among the nodes in S. Thus, i announces the sending of $\Delta/|S|$ to each node $j \in S$ and reserves $d_i/|S|$ in the buffer b_j. This reserved amount is needed to ensure that the bounded step rule is not violated in the asynchronous case, as shown in the analysis below.

Let r_j be the announced load received from node j, that is, the amount that j wants to send to i. In the `RECEIVE` part, the total announced load $R^o = \sum_{j\in N(i)} r_j$ is used to calculate the amount of the load that i will refuse. Hence, for each $j \in N(i)$, i refuses $\max\left(0,\ r_j\left(1 - \frac{\varepsilon d_i}{R^o}\right)\right)$ of the announced load of node j. Note that if i receives more than $\varepsilon\ell_i$, the bounded step rule is violated. Thus, if we have an *excess* of announced load, that is, $R^o - \varepsilon\ell_i > 0$, then the refused amount is greater than zero in order to respect the bounded step rule.

Algorithm 3 formalizes the above discussion.

```
begin
    /* SEND part                                                    */
    S ← {j ∈ N(i) : (1 + ε)^3 ℓ_j < ℓ_i, b_j = 0}
    d_i ← p_i − Σ_{j∈N(i)} b_j
    Δ ← (ε/(1+ε)) d_i
    foreach j ∈ S do
        Announces the sending of Δ/|S| to j
        b_j ← d_i/|S|
    /* RECEIVE part                                                 */
    R^o ← Σ_{j∈N(i)} r_j the total received announcements
    foreach j ∈ N(i) do
        Refuse max(0, r_j (1 − ε d_i / R^o)) of the received announcement of node
        j
        r_j ← 0
end
```

Algorithm 3: `SEND-RECEIVE` algorithm for player i

Algorithm 4 describes the procedure `LISTENER`, which has maximum execution priority. Here `LISTENER` is responsible for the real load transfer between nodes, and it runs every time the node receives a message of a refused load, even if the refused load is equal to zero. The procedure `LISTENER` has the highest execution priority of the node's tasks, and is thus immediately run when the node receives a refused load message.

```
begin
    Let c be the refused load received from node j:

     − send b_j − c to node j

     − b_j = 0    /* no load is now reserved to node j */

end
```

Algorithm 4: `LISTENER` algorithm for player i

When `SEND-RECEIVE` is running the `RECEIVE` part, messages of refused load that are generated trigger the neighbors' `LISTENER` procedures, which then start sending load to this node. We assume that `SEND-RECEIVE` and `LISTENER` have *atomic execution*, that is, once their execution starts on a node, it cannot be interrupted by other tasks on the same node.

Lemma 8. *Algorithm `SEND-RECEIVE` respects the inertia and bounded step rules.*

Proof. The inertia rule is clearly respected by the selection of the set S in the first line of the `SEND-RECEIVE` algorithm. Since `SEND-RECEIVE` does not send load, we only need to show that load receiving does not violate the bounded step rule. Let p_i^t and $p_i^{t'}$ be, respectively, the values of p_i immediately before and after the execution of the `LISTENER` algorithm. Therefore, we need to show that $p_i^{t'} \leq (1+\varepsilon)p_i^t$.

Note that if $R^o < \varepsilon d_i$, then the node refuses 0 of the announced load, and therefore receives all load R^o. Otherwise, the node refuses $\left(1 - \frac{\varepsilon d_i}{R^o}\right) \sum_{j \in N(i)} r_j = R^o - \varepsilon d_i$, and therefore receives load εd_i. Thus, in both cases, the node receives at most $\varepsilon d_i \leq \varepsilon p_i$. □

Lemma 9. *Algorithm `LISTENER` respects the inertia and bounded step rules.*

Proof. Let p_i^t and $p_i^{t'}$ be, respectively, the values of p_i immediately before and after `LISTENER` is run. During the execution of `LISTENER` , node i does not receive load; thus, trivially, $p_i^{t'} \leq (1+\varepsilon)p_i^t$. In addition, `LISTENER` sends load to a node chosen by `SEND-RECEIVE`, and hence, by Lemma 8, it respects the inertia rule. Therefore, we need to show that $\frac{1}{1+\varepsilon}p_i^t \leq p_i^{t'}$.

Note that the sent load is equal to at most the load that had been announced for sending, and this announced load is at most $\varepsilon/(1+\varepsilon)$ of the load that was reserved (see the two statements within the loop of the `SEND` part). Thus, after sending, the node has at least $1/(1+\varepsilon)$ of the reserved load. Since the reserved load is part of the node's load, after `LISTENER` is run, the node has at least $1/(1+\varepsilon)$ of the previous load. □

Theorem 10. *In asynchronous and heterogeneous networks, if the nodes use the bounded step method described in algorithms 3 and 4, the system converges to an ε-Nash equilibrium.*

Proof. We show that if the system is not in a Nash equilibrium, then the unsatisfied nodes send a non-null load amount to their neighbors. This fact, together with Theorem 7 and lemmas 8 and 9, proves the result.

Consider a non-Nash equilibrium state and let i be an unsatisfied node. When i executes `SEND-RECEIVE`, we have two cases. If $\Delta = 0$, then $\sum_{j \in N(i)} b_j \neq 0$. Therefore, i has a reserved load that will be sent to its neighbors. Otherwise, if $\Delta \neq 0$, i reserves the load that will be sent to at least one neighbor. In both cases, the neighbors never refuse all the announced load of node i; that is, i will always send a portion of its load to the selected neighbors. □

4.3 Aggressive Method

This section describes the *aggressive method*. It is thus named because it sends load without worrying about high load fluctuations or accumulations, it does not use much information about its neighborhood, and it prioritizes neighbors with low load when sending load. Since we want to aggressively distribute the load, the load is sent using the *water-fill* algorithm because, among the techniques discussed, it is the most sophisticated: While it prioritizes less loaded nodes, it also performs load balancing on participating nodes. In short, the aggressive method sends large load amounts without worrying about the negative consequences, hoping to quickly converge to a Nash equilibrium.

To reach a Nash equilibrium, the loads should be sent to lighter nodes. Thus, in the aggressive method, neighbors with higher loads are not considered by the algorithm, unlike the egalitarian method, where heavily loaded nodes are considered to compute the load equilibrium value and hence avoid problems of load returning, load accumulation, and high load fluctuations in the nodes. The set S is defined as $S = \{j \in N(i) : (1+\varepsilon)^3 \ell_j < \ell_i\}$, and Δ is equal to the excess of processing weights in i regarding the mean of $S \cup \{i\}$. Thus, unless $S = \emptyset$, Δ is always positive, which demonstrates the aggressiveness in sending the load. Algorithm 5 describes the method.

begin
  Let $S = \{j \in N(i) : (1+\varepsilon)^3 \ell_j < \ell_i\}$
  Let $\overline{x} = \frac{p_i + \sum_{j \in S} p_j}{s_i + \sum_{j \in S} s_j}$
  Let $\Delta = (\ell_i - \overline{x}) s_i$
  `water-fill`(Δ, S)
end

Algorithm 5: Aggressive method algorithm for player i

The `water-fill()` algorithm used in Algorithm 5 is a straightforward generalization of Algorithm 1 for the case of heterogeneous players; therefore the generalization details are omitted.

Claim 11. *Let T, L, and L' be finite sets of real numbers, where L and L' have the same cardinality. If the sorted vector of the values in L' is lexicographically greater than the sorted vector of the values in L, then the sorted vector of the values in $T \cup L'$ is lexicographically greater than the sorted vector of the values in $T \cup L$.*

Theorem 12 Convergence. *In asynchronous and heterogeneous networks, if the nodes use the aggressive method described in Algorithm 5, then the system converges to an ε-Nash equilibrium.*

Proof. In the asynchronous case, only a subset of nodes execute the algorithm at a given time instant. Let $A \subseteq V$ be the set of players executing Algorithm 5 or have its load changed (by one of the players executing the algorithm) at time t. Let L_t be the load vector of the nodes in A at time t sorted in non-decreasing order of loads. We show that L_{t+1} is lexicographically greater than L_t. Let $A_{\min} \subseteq A$ be the set of nodes that have the lowest load among the nodes of A. At least one node $u \in A_{\min}$ is adjacent to a node v such that $\ell_v^t > \ell_u^t$. At time t, using Algorithm 5, u does not send load and v sends part of its load to u. Let $\overline{x}$ be the mean load of node v, computed by Algorithm 5. At time $t+1$, we have $\ell_v^{t+1} \geq \overline{x} > \ell_u^t$. Therefore, the load of v (or any other node in $A \backslash A_{\min}$) decreases but never becomes smaller than the load u had in the previous step. Thus, L_{t+1} is lexicographically greater than L_t. Therefore, using Claim 11, the load vector of the nodes in V at time t sorted in non-decreasing order is lexicographically greater than the sorted load vector of the nodes in V at time $t+1$. □

5 Empirical Analysis

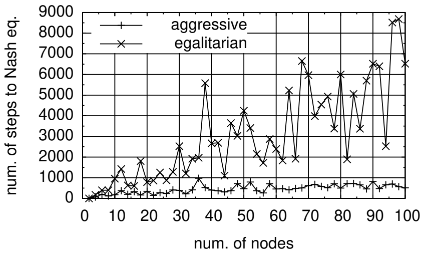
Next, we empirically compare the algorithms proposed in Section 4. To do this, we consider networks with properties similar to those of real-world networks. The comparison metric is the number of steps to reach an ε-Nash equilibrium, and we change important parameters of the simulation to analyze their effect on the algorithms' performance.

Unstructured network topologies, given by random networks are natural models to perform comparisons. In this context, the most commonly studied models are the Erdős-Renyi model $G(n,p)$ [Erdös and Rényi, 1959] and the closely related model $G(n,M)$. Neither model is consistent with the properties observed in unstructured real-world networks. An alternative, which we use in the simulations, is the *scale-free networks* model [Barabási and Albert, 1999]. Scale-free networks are noteworthy because many empirically observed networks appear to be scale free [Barabási and Albert, 1999], including computer networks, social networks, transportation networks, and citation networks. In the case of *structured* networks (e.g., some P2P networks [Ratnasamy et al., 2001, Rowstron and Druschel, 2001, Stoica et al., 2001]) or parallel systems, the hypercube network is a useful evaluation model. Through brute force simulations in small networks, we observed that the worst case topology is the linear (or path) network, which is a network with nodes $V = \{v_1, v_2, \ldots, v_n\}$, where the edges are of the form (v_i, v_{i+1}) for $i = 1, \ldots, n-1$.

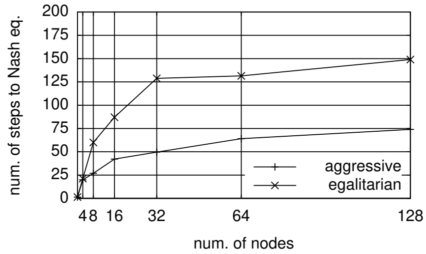
In the simulations, a *network configuration* consists of the network's size, topology, node loads, and node speeds and whether execution is asynchronous or completely parallel. Thus, the results presented for a given network configuration are obtained from the arithmetic mean of 10 executions on the same configuration. It is worth noting that if some parameters are randomly generated, then the number of steps needed in each run of the "same" configuration can change (e.g., the simulation is performed 10 times in a scale-free network with n nodes and taking the arithmetic mean, but in each single run the generated network may be different).

Let $s_{\max}$ be the greatest speed among all the nodes of an instance. To simulate the asynchronous system, at a given time a node i runs its algorithm with probability $s_i/s_{\max}$. This probability is justified by the fact that the fastest (slowest) nodes, that is, those with higher (lower) processing power, run their algorithms more (less) often. In all simulations, we use $\varepsilon = 0.001$ and networks with size $2, 4, 6, \ldots, 98, 100$, except for the hypercube network, where the evaluated sizes are $2, 4, 8, 16, 32, 64$, and 128.

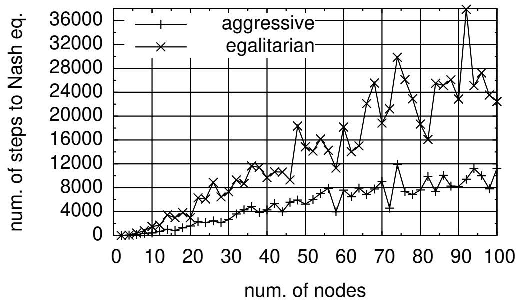
Figure 4 shows the results for the scale-free, hypercube, and path topologies. The processing weights and speeds were chosen uniformly at random in the ranges $[1, 1000]$ and $[1, 10]$, respectively, and the network is asynchronous. Due to the great difference between the results of the bounded step and the other two methods, we present the results in separate charts.



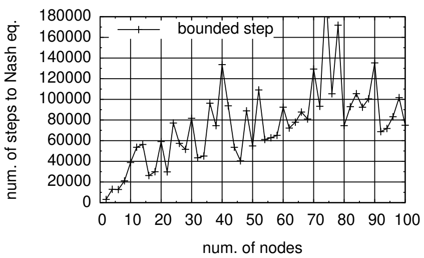
(a) Scale-free



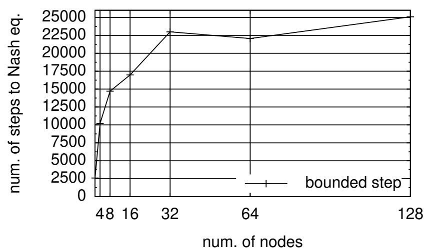
(b) Hypercube



(c) Path



(d) Scale-free



(e) Hypercube

Figure 4: Simulations on asynchronous networks

In Figure 4, we note that the bounded step method has a convergence time much greater than for the other methods. Simulation of the path network topology using the bounded step method became unfeasible for some network sizes due to long simulation times; thus, we choose not to present these results here. Comparing the results of Figures 4(a), 4(b) and 4(c), we note that in all cases the aggressive method reaches a Nash equilibrium more quickly. Moreover, in all methods the number of steps is highly dependent on the network topology used, with the path topology being the worst. Examining the path topology more closely, we can see that a load at one end of the path must pass through all the nodes to reach the other end. Thus, we can suspect that the number of steps needed to reach a Nash equilibrium is highly influenced by the network diameter. This does not, however, seem to be true in general, since most scale-free networks have a diameter on the order of $\ln \ln n$ [Cohen and Havlin, 2003] and hypercubes have a diameter $\log n$, but the simulations shows the hypercube has faster convergence.

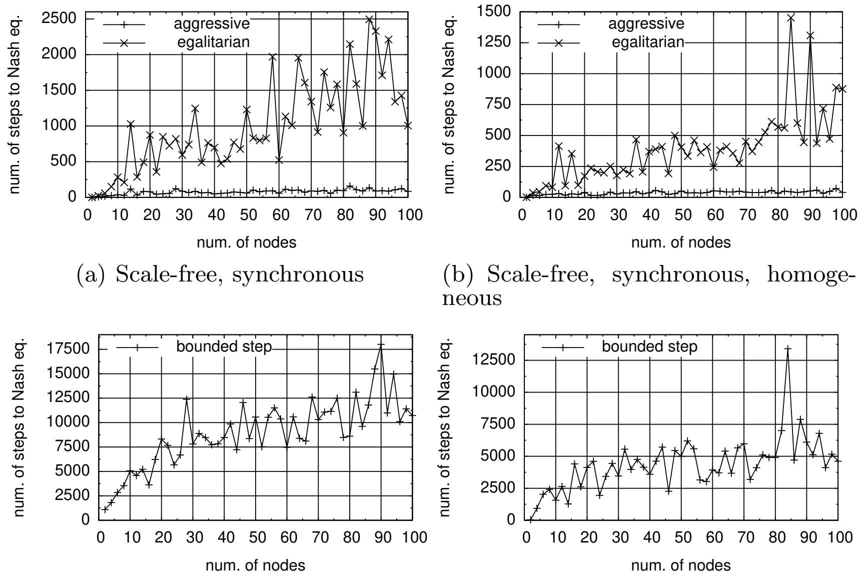
In Figure 5, we analyze the impact of asynchrony and heterogeneity. These results can be compared with those of Figure 4. Figures 5(a) and 5(c) present the simulation results for the case where all players perform in parallel, that is, the system is synchronous. Figures 5(b) and 5(d) examine the synchronous and homogeneous cases, that is, where all the nodes have the same speed. To make a more accurate comparison, we set all nodes to have a speed equal to 5.5, which is the average speed in the simulations of Figure 4.

We can see that in the synchronous case, the convergence becomes three to five times faster for the aggressive and egalitarian methods, and about 10 times faster for the bounded step method. The bounded step method is the method that most benefits from synchronicity due to the exchange of messages, because a node that executes its algorithm more often does not require waiting for a reply message from a node that executes its algorithm less frequently, as in the asynchronous case. When the synchronous and homogeneous case is considered, the convergence becomes about 1.5 times faster in all methods, compared with the synchronous and heterogeneous case.

In short, we conclude the aggressive method to be the most efficient, while the bounded step method proves to be an unfeasible alternative due to the complexity of its implementation and convergence time.

6 Conclusion and Future Work

This article discusses the problem of load balancing in a network where a central monitoring system is impractical or impossible to implement, leading to nodes (players) that act in a game-theorical fashion. We show that the network does not stabilize if rules are not imposed. Thus, rules are created with the condition that



(a) Scale-free, synchronous

(b) Scale-free, synchronous, homogeneous

(c) Scale-free, synchronous

(d) Scale-free, synchronous, homogeneous

Figure 5: Simulations on synchronous networks

they must be simple, distributed, and meet the players' objectives. We discuss how the rules can be created and we present three sets of rules: aggressive, egalitarian, and bounded step. We present a formal proof of the correctness of each set of rules . To evaluate the performance of each set of rules, we perform simulations that empirically show that the aggressive method is the most efficient one, while the bounded step method becomes unfeasible in practice.

From a theoretical point of view, it would be interesting for future work to show lower and upper bounds for the number of steps in each method, possibly considering specific topologies (e.g., the path topology). From a practical standpoint, one can create more efficient methods, analyze the case of transient populations, and examine whether these methods could be used in similar models of network load balancing.

Another important future direction is to consider that the nodes can lie about their loads. In this case, the nodes behave selfishly because they can, for example, say that they are overloaded and then it is possible to cheat by using the algorithms proposed in this paper. In this paradigm, mechanism design techniques must be used to design rules that do not encourage users to lie about their load.

Acknowledgements

We would like to thank the brazilian financial agencies CNPQ (under processes numbers 306860/2010-4, 473867/2010-9, 477692/2012-5) and FAPESP (under process number 2009/15008-1).

References

[Awerbuch et al., 2008] Awerbuch, B., Azar, Y., and Khandekar, R., "Fast load balancing via bounded best response";. In *SODA '08: Proceedings of the 19th Annual ACM-SIAM Symposium on Discrete Algorithms* (2008). pages 314–322, Philadelphia, PA, USA.

[Barabási and Albert, 1999] Barabási, A.-L. and Albert, R., "Emergence of scaling in random networks";. *Science* (1999), 286, pp. 509–512.

[Berenbrink et al., 2006] Berenbrink, P., Friedetzky, T., Goldberg, L. A., Goldberg, P., Hu, Z., and Martin, R., "Distributed selfish load balancing";. In *SODA '06: Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms* (2006). pages 354–363, New York, NY, USA.

[Cohen, 2003] Cohen, B., "Incentives build robustness in BitTorrent";. In *Proceedings of the First Workshop on Economics of Peer-to-Peer Systems* (2003).

[Cohen and Havlin, 2003] Cohen, R. and Havlin, S., "Scale-free networks are ultrasmall";. *Physical Review Letters* (2003), 90(5).

[CondorTeam, 2010] CondorTeam (accessed April 2, 2010). "Condor DAGMan";. http://www.cs.wisc.edu/condor/dagman/.

[Erdös and Rényi, 1959] Erdös, P. and Rényi, A., "On random graphs, I";. *Publicationes Mathematicae (Debrecen)* (1959), 6, pp. 290–297.

[Even-Dar et al., 2007] Even-Dar, E., Kesselman, A., and Mansour, Y., "Convergence time to Nash equilibrium in load balancing";. *ACM Transactions on Algorithms* (2007), 3(3), pp. Article 32.

[Even-Dar and Mansour, 2005] Even-Dar, E. and Mansour, Y., "Fast convergence of selfish rerouting";. In *SODA '05: Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms* (2005). pages 772–781, Philadelphia, PA, USA.

[Feldman et al., 2004] Feldman, M., Lai, K., Stoica, I., and Chuang, J., "Robust incentive techniques for peer-to-peer networks";. In *Proceedings of the EC '04* (2004). pages 102–111.

[Foster et al., 2001] Foster, I., Kesselman, C., and Tuecke, S., "The anatomy of the Grid: enabling scalable virtual organization";. *International Journal of High Performance Computing Applications* (2001), 15(3), pp. 200–222.

[Goldberg, 2004] Goldberg, P. W., "Bounds for the convergence rate of randomized local search in a multiplayer load-balancing game";. In *PODC '04: Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing* (2004). pages 131–140, New York, NY, USA.

[Golle et al., 2001] Golle, P., Leyton-Brown, K., Mironov, I., and Lillibridge, M., "Incentives for sharing in peer-to-peer networks";. In *Proceedings of the WELCOM '01* (2001). pages 75–87.

[Hayes, 2008] Hayes, B., "Cloud computing";. *Communications of the ACM* (2008), 51(7), pp. 9–11.

[Heckmann et al., 2005] Heckmann, O., Liebau, N., Darlagiannis, V., Bock, A., Mauthe, A., and Steinmetz, R., "A peer-to-peer content distribution network";. In *Lecture Notes in Computer Science* (2005). pages 69–78.

[Kim and Kameda, 1992] Kim, C. and Kameda, H., "An algorithm for optimal static load balancing in distributed computer systems";. *IEEE Transactions on Computers* (1992), 41(3), pp. 381–384.

[Lin and Raghavendra, 1992] Lin, H.-C. and Raghavendra, C. S., "A Dynamic Load-Balancing Policy with a Central Job Dispatcher (LBC)";. *IEEE Transactions on Software Engineering* (1992), 18(2), pp. 148–158.

[Lu et al., 2006] Lu, K., Subrata, R., and Zomaya, A. Y., "Towards decentralized load balancing in a computational grid environment";. In *Advances in Grid and Pervasive Computing, First International Conference, GPC 2006, Taichung, Taiwan* (2006). pages 466–477.

[Nash, 1951] Nash, J., "Non-cooperative games";. *The Annals of Mathematics* (1951), 54(2), pp. 286–295.

[Papadimitriou, 2001] Papadimitriou, C., "Algorithms, games, and the internet";. In *STOC '01: Proceedings of the Thirty-Third Annual ACM Symposium on Theory of Computing* (2001). pages 749–753, New York, NY, USA.

[Ratnasamy et al., 2001] Ratnasamy, S., Francis, P., Handley, M., Karp, R., and Schenker, S., "A scalable content-addressable network";. In *Proceedings of the SIGCOMM '01* (2001). pages 161–172, New York, NY, USA.

[Riska et al., 2000] Riska, A., Smirni, E., and Ciardo, G., "Analytic modeling of load balancing policies for tasks with heavy-tailed distributions";. In *WOSP '00: Proceedings of the Second International Workshop on Software and Performance* (2000). pages 147–157, New York, NY, USA.

[Rowstron and Druschel, 2001] Rowstron, A. and Druschel, P., "Pastry: scalable, decentralized object location, and routing for large-scale peer-to-peer systems";. *Lecture Notes in Computer Science* (2001), 2218, pp. 329–350.

[Stoica et al., 2001] Stoica, I., Morris, R., Karger, D., Kaashoek, F., and Balakrishnan, H., "Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications";. In *Proceedings of the SIGCOMM '01* (2001). pages 149–160, New York, NY, USA.

[Subrata et al., 2008] Subrata, R., Zomaya, A. Y., and Landfeldt, B., "Game-theoretic approach for load balancing in computational grids";. *IEEE Transactions on Parallel and Distributed Systems* (2008), 19(1), pp. 66–76.