Power saving and load balancing in data centers

Index

Abstract

The importance of telecommunications in our society is rapidly increasing day by day. Because of this, the number of servers and data centers is increasing at a very high rate, which entails a very important increment on the power consumed by them. Besides, the growing complexity of these systems makes managing more and more difficult.

To face the first problem, we study the behavior of the power consumption in a server depending on its utilization, which presents little variations when changing the utilization from 1% to 100%, but drops drastically when turning the servers off. Seeing this, we propose an algorithm that will reallocate the load of the servers in a data center in order to gather it in the less possible number of servers, so the rest can be turned off.

With this algorithm we manage to get high saving values, but the execution time in the system increases very much, so this is not enough. To solve this situation, we propose another algorithm to balance the load in the servers in order to reduce the execution time and keep it in a reasonable interval.

To cope with the problem of complex management, the solutions proposed in this document are developed in an autonomic way: each server interacts with a small number of neighbors and acts at a local level, avoiding the need for a centralized control of the system.

We test the proposed algorithms in two different scenarios: *distributed server farm* and *client-server*.

On the distributed server farm scenario, where we just consider the servers (not their relationships with the clients), we get a maximum power saving of 11.66%. On the other hand, on the client-server model (where some complexity is added by considering not only the servers, but also their connections with the clients) the savings achieved rise up to 19.56%, thanks to an enhancement of the algorithms used in the first scenario.

1. Introduction

1.1 Context

Telco and internet networks have become an indispensable element in our digital society. Nowadays, data centers are required in any sector of the economy. That is why the number of servers and data centers is increasing everywhere at a very high rate. This increment has obviously an impact on the energy consumption.

1.2 The problem

A report [1] developed by the United States (U.S.) *Environmental Protection Agency* (EPA) estimates that, in 2006, servers and data centers used about 61 billion kwatt-hours (kWh). This value represents 1.5% of total U.S. energy consumption. Translating this value into money means talking of an electricity expense of $4.5 billion!

The power consumption in 2006 more than doubled that in 2000, and it is estimated that by 2011 this value could be doubled again. This is why concerns about the amount of energy consumed by data centers are increasing day by day.

1.3 Objective of the thesis

The results showed in the report mentioned in *1.2* were the main motivation for the work carried out in this thesis. We wanted to change that situation, or at least help in reducing the intensive increase in power consumption.

Considering a data center as a set of distributed servers, the objective of this thesis is to provide those servers with a series of algorithms, based on the principles of self-organization, which would lead to an optimization of the power consumed in the system.

We will start defining the rules we should apply in the servers. After that, we will implement those rules in the model. Finally, we will run different simulations to test our rules and see if and how they improve the performance of the system.

1.4 Technical approach

We consider a system composed by a set of servers connected in an overlay. All of them have the same behavior, i.e. implement the same algorithms.

Two main algorithms are developed in this thesis: *Standby Algorithm* and *Overload Algorithm*.

Standby Algorithm is responsible for saving some energy in the system. Nodes exchange information with their neighbors and try to redistribute the load in a way that is possible for the system to reduce power consumption.

Overload Algorithm is responsible for reducing execution time in the system. Nodes with a high utilization ask for help to their neighbors, so they can reduce their load. With this algorithm the load present in the system gets distributed in order to achieve the expected execution time values.

To test how the system behaves in the presence of our algorithms we have done a series of simulations, executed in *PeerSim* simulator (which will be explained in *Section 5*).

The classes implementing our algorithms have been developed in Java, since it is the programming language required by *PeerSim*.

1.5 Structure of the thesis

First, right after the introduction, we present in *Section 2* the technical background for our project.

Next, we explain in detail the problem we are facing and how we try to solve it in *Section 3*.

After that, in *Section 4* we find the technical approach used, including the description of the system architecture and the definition of the algorithms executed in each of the cases analyzed (energy saving in distributed server farm, load balancing in client-server model, load balancing and power saving in client-server model).

In *Section 5* we first introduce the simulator we used, *PeerSim*, and explain its main features. Then, we explain the code of the different Java classes created for modeling the system in the different cases considered.

Later, we show and analyze the results of the simulations in *Section 6*.

Lastly, in *Section 7* we compare the obtained results with those obtained by using other solutions; and in *Section 8* we present some conclusions and talk about possible future work.

To conclude, we include the references to the works and papers we consulted for the elaboration and redaction of this thesis in *Section 9*.

2. Technical background

2.1 Autonomic Computing and Communications

Advances in IT and ICT have resulted in an exponential growth in computing systems and applications that impact all aspects of life. Nevertheless, scale and complexity of such systems and applications represent obstacles to further developments. Configuration, healing, optimization, protection and, in general, management challenges are beginning to overwhelm the capabilities of existing tools and methodologies, and rapidly render the systems and applications almost unmanageable, not optimized and insecure [2].

In 2001, IBM introduced the *Autonomic Computing* initiative [3] with the aim of developing self-managing systems. The word autonomic is inspired by the functioning of the human nervous system and is aimed at designing and building systems that present self-configuration, self-healing, self-optimization and self-protection features.

The human body's autonomic nervous system is the part of the nervous system that controls the vegetative functions of the body, such as circulation of the blood, heart rate, body temperature, production of chemical *messengers* (i.e. hormones), etc.; thus hiding from the conscious brain the burden of dealing with these and many other low-level, yet vital, functions.

In a similar way, autonomic computing systems take decisions, according to high-level policies defined by operators, self-managing computer resources in order to hide complexity from operators and users. They will constantly check their environment and adapt to changing conditions.

The concept of *Autonomic Computing* has been extended to *Autonomic Communications* to indicate Network and Service Frameworks capable of creating, executing and providing communication and content services in an autonomic way, i.e. hiding complexities to humans as showing self-* behaviors.

Indeed, this approach would meet the needs of: enabling integration and convergence of ICT and Telco solutions hiding complexities to Providers; reducing capital and operative expenses and human errors in managing such infrastructures; fostering a new paradigm of services meeting, and even anticipating, the customers' needs more flexibly.

Autonomic systems are typically distributed, complex and concurrent, and are composed of multiple interacting Autonomic Components (AC). An AC can be modeled in terms of two main control loops with sensors (for self-monitoring), effectors (for self-adjustment), knowledge and planner/adapter for exploiting the policies:

- *Local control loop* for self-awareness, internal management and recovery from faults.
- *Global control loop* for environment awareness, allowing changing behavior and even environment (through communication with other elements).

Figure 1 shows an example of Autonomic Component as developed by the IST Project CASCADAS [4], whose objective is to develop and demonstrate a prototype of Service Ecosystem for dynamic composition and execution of autonomic services.

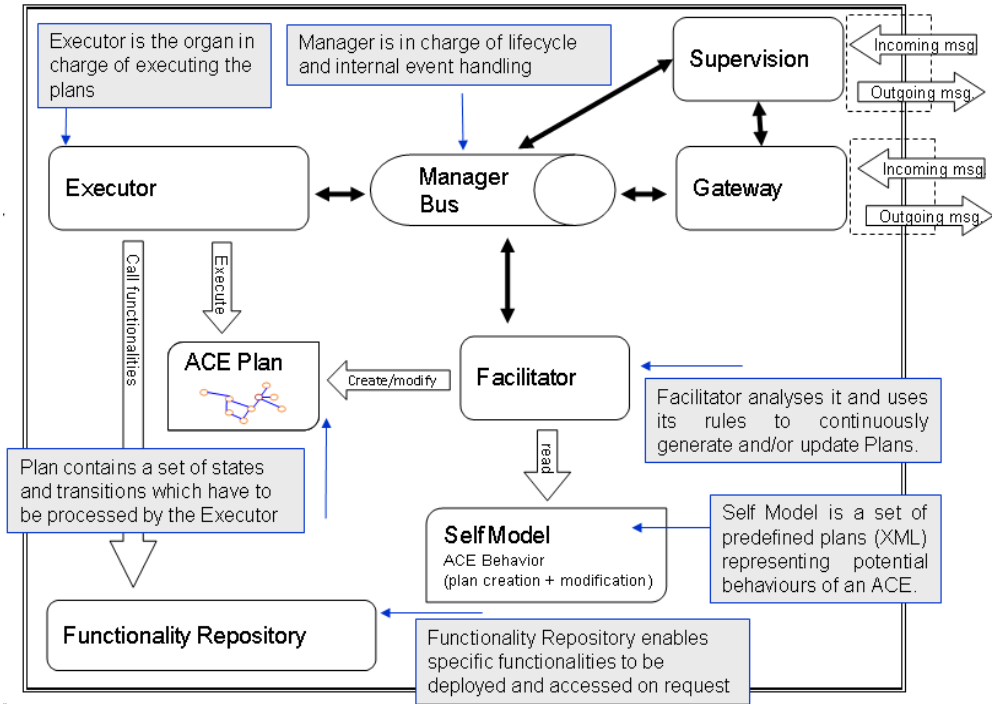


Figure 1: Autonomic Component defined in CASCADAS

Upon launching the *Autonomic Computing* initiative, IBM defined four general properties a system should have to constitute self-management: self-configuration, self-healing, self-optimization and self-protection. Since the launch of *Autonomic Computing*, the self-* list of properties has grown substantially, now including features such as self-anticipation, self-adaptation, self-definition, self-destruction, self-diagnosis, self-organization, self-recovery, etc.

2.2 Self-organization

A key challenge of the autonomic computing initiative has been to draw upon self-* properties in systems other than computational ones in order to develop new computing systems. Biology has been a key source of inspiration. Group-living animals have provided inspiration for the field of collective, or swarm intelligence [5] which models problems through the interactions of a collection of agents cooperating to achieve a common goal [2, 6].

The mechanisms that lead to self-organization in biological systems differ from those occurring in physical systems in that they are influenced by biological evolution. Thus there is the possibility of using algorithms inspired by biological evolution. These are part of what has been described as biologically-inspired or *nature-inspired* computing. A diverse range of fields have been analyzed, including evolution and genetics, bacterial adaptive mechanisms, morphogenesis and self-organizing principles more broadly [7].

Another aspect of the interest in self-organization has been the development of interest in complex networks [8] as an application area for computer science. The reason for the extraordinary interest of the computer science community in complex networks is arguably two-fold: first, the re-discovery of *small world* effects by Duncan Watts and Steven Strogatz and second, the discovery of the *scale-free* properties of many man-made physical networks (e.g. the Internet) and overlays (e.g. the World-Wide-Web).

The first experimental study of the *small world* effect was conducted by Milgram in the late 1960s [9] and demonstrated that complete strangers are often linked by a relatively short chain of

acquaintances. The newfound popularity of the concept is attributable to the efforts of Watts and Strogatz to formalize it using graph theory [10, 11], which led to associating *small world* topological properties (e.g. logarithmic relationship between size and diameter) with efficiency in communication networks [12].

After Faloutsos et al. published their now famous paper [13] presenting experimental evidence that the Internet exhibited *scale-free* properties, the community realized that the work on random graphs initiated by Erdos and Renyi in 1960 [14] was a powerful tool to understand the dynamics of the Internet. Since then there has been much research effort on understanding the nature of the self-organizing complexity that arises from use of the Internet, e.g., [15].

Further interest in developing self-organizing systems has considered the role that decentralized control of distributed systems can play in self-organization. Peer-to-peer networks can be used as a basis for self-organization among elements, in order to develop complex adaptive systems [16, 17] or self-organizing multi-agent systems [18, 19]. These draw upon the emergence of novel properties at the whole system level when many elements are brought together automatically, without being programmed in by system developers, and inspiration is drawn from biological systems.

Babaoglu and colleagues have developed a number of systems in this area [16, 17], all of which draw upon a dynamic network of peers with some sort of adaptive agents interacting. Babaoglu et al. [17] review these models and a variety of others all looking at how to manage systems running on unpredictable network environments drawing upon biological inspiration.

Another important area of investigation of self-organization has focused upon how individuals can cooperate to produce self-organizing behavior. Like other aspects of the analysis of self-organization, this has been inspired by cooperation in nature, in systems ranging from microorganisms [20] to human beings [21].

Overall the concept of self-organization has motivated a great deal of research that promises to have considerable impact on emerging autonomic computing and communications technologies.

In order for the distributed system to be able to adapt to a change in the overall demand (assuming that the total processing capability is sufficient), it is necessary that resources can be repurposed and re-allocated without centralized management functionalities in order to obtain self-configuration and scalability.

This *self-(re)allocation* of individual resources, based on locally available information and local interactions, is called *differentiation* because it shares many characteristics with the eponymous phenomenon in biological morphogenesis. It is easy to see the self-differentiation process as a particular way of self-organization.

The question of differentiation at all levels from cells to organisms, social systems or ecosystems, is ubiquitous in biology. This vast issue can be subdivided according to the context of differentiation. Here we focus on systems where identical (or nearly identical) individuals differentiate through interactions among themselves and/or with their environment, a problem that has been addressed many times in theoretical biology. One current vision of such differentiation phenomena is that the system (or the individuals) can exist in multiple states between which individuals (or the whole system) can switch.

As the autonomic component's behavior is represented as a series of states, it is realistic to consider the transition between the different states as a *self-organization* process.

Our models are based on the assumption that in an autonomic system there are several nodes providing the same functionality. This way, it is possible for a server to undertake the tasks of a neighbor.

The use case we will analyze in the next sections is about the self-optimization of energy consumption of Telco and Internet data centers by minimizing the number of running servers, i.e. maximizing the number of servers in *standby* state by redistributing the tasks to be executed. The approach we followed consists on aiming at exploiting local interactions (interactions among neighbor servers in an overlay network) and self-organization algorithms.

2.3 Matchmaker algorithms

To achieve the self-organization we mentioned previously we need an exchange of information in the system. This exchange consists basically on the utilization level of the different nodes.

As we are aiming for a decentralized system, composed by autonomic elements, we cannot have a control element gathering the information of the states of the nodes and creating the matches: the exchange has to be done at a local level. We want any of the elements in the system to be able to carry out the exchange, so any of the servers has to be able to act as a matchmaker.

Fabrice Saffre et al. present in [22] a matchmaker algorithm that is used to achieve self-organization in an autonomic system. Through this algorithm, nodes cluster depending on their type. Three agents are involved in the algorithm: the *initiator*, the *matchmaker* itself and the *candidate*. The initiator contacts with one of its neighbors randomly, which becomes the matchmaker, and sends the information of the requested type of node. The matchmaker looks for a *candidate* among their neighbors and returns that information to the *initiator*. The *initiator* will then try to connect to the *candidate*.

The matchmaker algorithm we use to achieve self-optimization in our system is very similar to the one presented by Saffre, but we do not include the role of the *initiator*. Instead, we have the *matchmaker* and two *candidates*. The way the algorithm starts is also a bit different. While in [22] the *initiator* starts with the algorithm, in our case that corresponds to the *matchmaker*. At a random time, a server can *wake up* as a *matchmaker*. When that happens, the *matchmaker* takes two random neighbors (*candidates*) and compares their load. After that, if the *matchmaker* considers that any load exchange can be done to increase power saving it notifies the *candidates*, which will carry out the load transfer.

2.4 CASCADAS

Today's Internet is rapidly evolving towards a collection of highly distributed, pervasive, communication-intensive services. In the next future, such services will be expected to (i) autonomously detect and organize the knowledge necessary to understand the general context in which they operate, and (ii) self-adapt and self-configure to get the best from any situation in order to meet the needs of diverse users, in diverse situations, without explicit human intervention. These features will enable a wide range of new activities that are simply not possible or impractical now. However, achievement of such capabilities requires a deep re-thinking of the current way of developing and deploying distributed systems and applications.

In this direction, a promising approach consists in conceiving services as part of an *open ecosystem* through which they can prosper and thrive at the service of users. This vision is attractive because it not only allows providing better services to end-users, but also meets the

emerging economic urge for service provision and system management deriving by the higher level of dynamism and variability of communication systems.

In this context, the *Component-ware for Autonomic Situation-aware Communications and Dynamically Adaptable Services* (*CASCADAS*) project is born. Its main goal is providing an autonomic component-based framework to support the deployment of a novel set of services through development of distributed applications capable of coping well with uncertain environments by dynamically adapting their plans as the environment changes in uncertain ways. *CASCADAS* vision is based on a set of complementary founding features that starts from state-of-the-art modern distributed computing and communication systems, for advancing towards autonomic and situation-aware communication services: context-awareness becomes situation-awareness; self-organization and self- adaptation converge into a concept of semantic self-organization; scalability assumes the form of self-similarity; modularity takes the form of a new autonomic component-ware paradigm that intrinsically features self-CHOP capabilities. *CASCADAS* is proving a robust and dynamic modular conceptual framework for building autonomic, self-organizing, semantic services, and act as high-level reference model for the production of a new generation of programmable communication elements that can be reused at different stack levels. Such component model, core of the framework, forms the fundamental software engineering abstraction of distributed self-similar components named Autonomic Communication Elements (ACEs).

Further information about *CASCADAS* can be found on the project's website [4].

3. Theoretical approach

3.1 Distributed server farms

A server farm is a collection of computer servers usually maintained by a company (e.g., an enterprise or a service provider) to accomplish server needs far beyond the capability of one machine. Server farms often have backup servers, which can take over the function of primary servers in the event of a primary server failure. Server farms are typically co-located with the network switches and/or routers which enable communication between the different parts of the cluster and the "clients" of the cluster.

In our work we treat with distributed server farms, opposite to the idea of the traditional data centers described before where all the servers are located in the same room or building [23].

We consider that the network is composed by a large set of nodes. Each one of them has a unique identifier and communicates with each other through message exchanges. The nodes are connected through an existing routed network, such as the Internet, where every node can potentially communicate with any other node.

Every node holds a list of identifiers of a set of nodes, which are considered its *neighbors*. This neighborhood relationship defines the topology of an *overlay network*. Given the large size and the dynamism of the considered system, neighborhoods are typically limited to small subsets of the entire network. The overlay topology can change dynamically, responding to changes in the environment as the incorporation and departure of nodes.

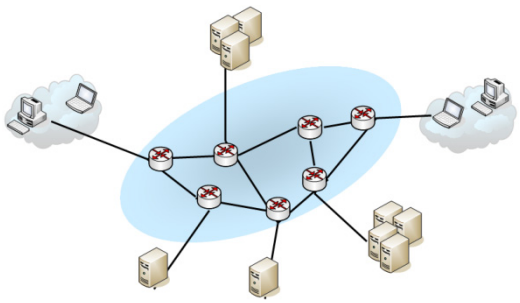


Figure 2: Distributed server farm

In the systems we study in this document we do not simulate dynamism, since we do not consider any changes in the number of nodes present in the system or the possibility of a node crashing and leaving the system. In the models we present, the overlay is created during the initialization stage and remains the same through all the simulation time.

3.2 Energy saving

In *Figure 3* (taken from [24]) we can see that the amount of energy consumed by a sleeping server is considerably smaller than that consumed by an idle server. On the contrary, the changes on the energy consumed by an active server when the utilization varies are not that big (the power consumption varies practically linearly).

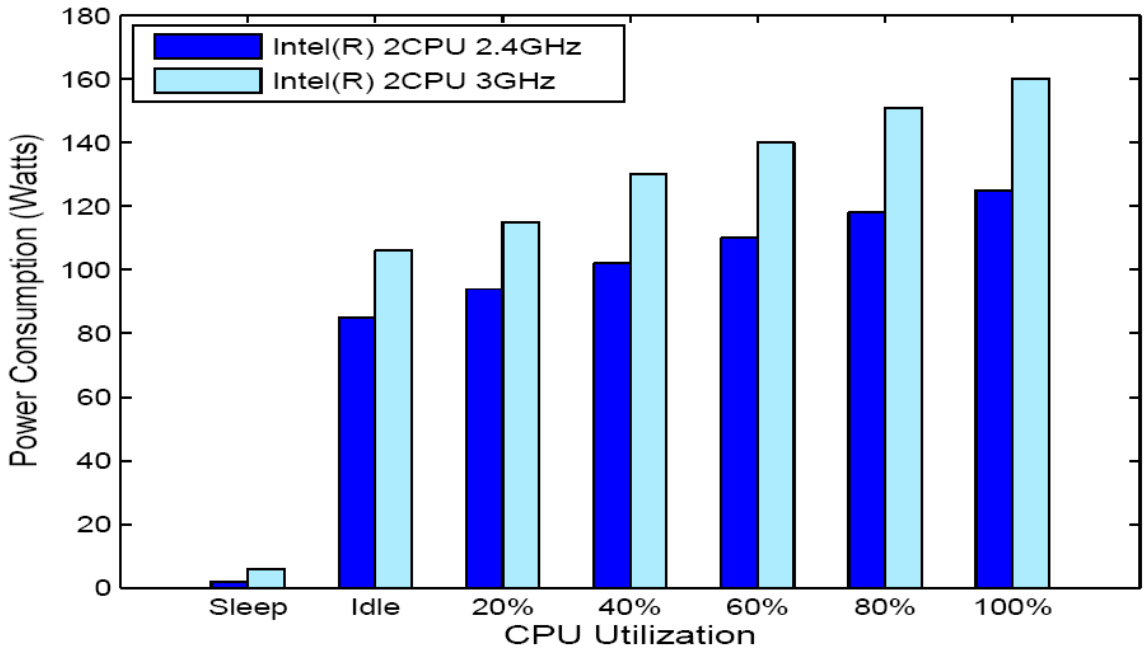


Figure 3: Power consumed by a server depending on CPU utilization

From this observation, having a group of servers with low utilization in a server farm could be seen as a waste of energy, since the same work could be executed by a smaller number of servers. The rest of the servers would be put to sleep (standby), reducing the energy consumption in the system.

This idea is explained through an example, where we consider a server farm composed by eight servers.

In *Figure 4* we see the situation we can find actually in most of the data centers. We have a number of servers that are *always* working. We do not pay any attention to their utilization and, of course, we do not do any optimization of the power consumed.

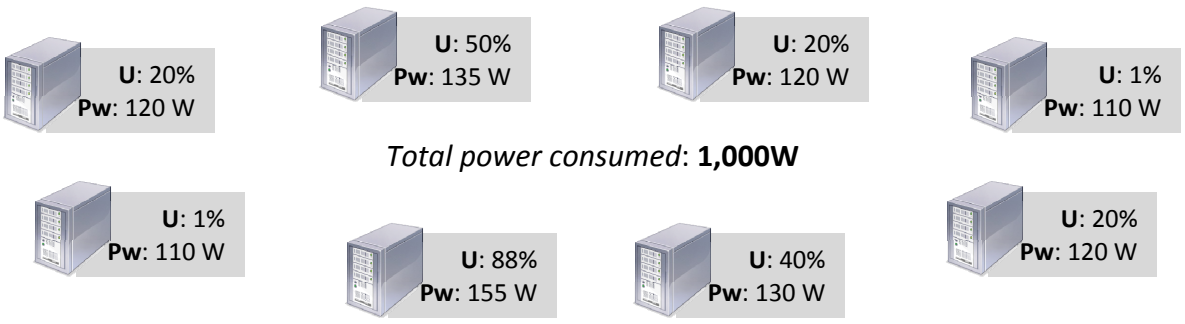


Figure 4: Actual situation

Considering what we said in the beginning of this section, in the system we find some servers with low utilization. The load executed by those servers could be transferred to other nodes to put some of the servers to sleep. This is shown in *Figure 5*, where we indicate with a lighter background the nodes with low utilization (considering an example threshold of 30%) and with arrows how their load could be transferred to another node.

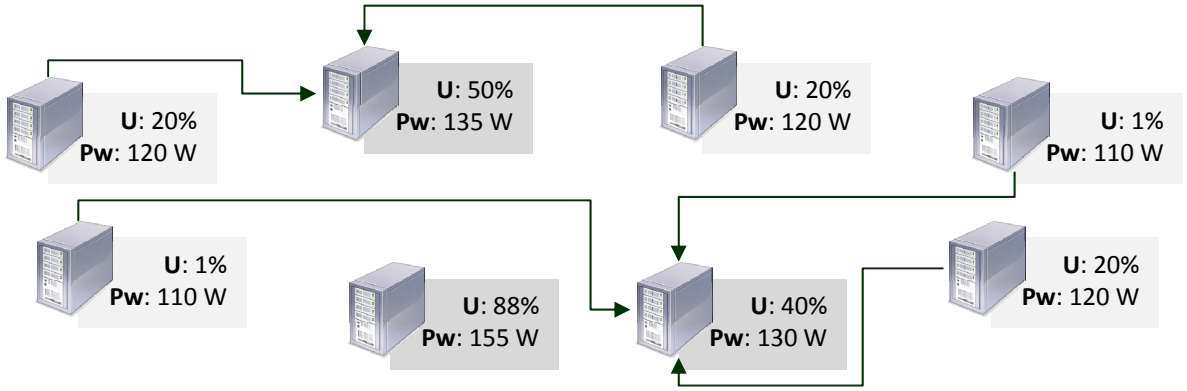


Figure 5: Proposed load transfer

Finally, in *Figure 6* we see how the system will look like after the load transfer proposed in *Figure 5*. We observe that now just three servers concentrate the load that was initially distributed among the eight of them. Their utilization is higher, and so is the power consumed by them. However, the remaining five nodes are sleeping (indicated with a *standby* sign), so their power consumption drops drastically.

In the end we have a power consumption of 475W, which means we achieved an energy saving of over 50% (the initial power consumption was 1,000W).

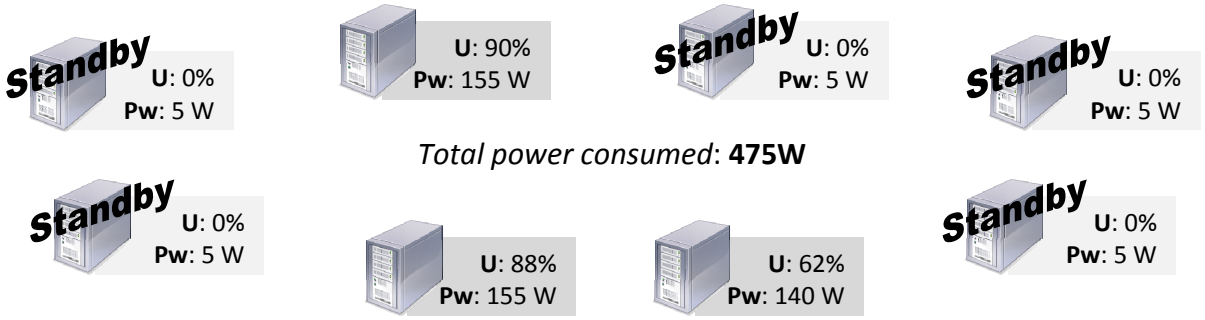


Figure 6: Situation after the actions proposed

The example illustrates, in a very simple way, that the assumptions presented in this section could lead to the achievement of some energy saving in data centers. It is also easy to see from the example the application of the principles of self-organization introduced in *2.2*: the nodes decide in an autonomic way how to distribute the load, the nodes that go standby and those who have to work.

3.3 Load balancing

In the previous section we describe a simple way to save energy in data centers, and we see through an example that said goal is achieved. Nevertheless, we do not pay any attention to the execution time in the system.

In *Figure 7* and *Figure 8* we show again the example presented in *2.2*, indicating now the execution time values in each sever and in the whole system. The execution time is calculated considering that the load of a server is the percentage of utilization (50% = 50 tasks) and that a

node can execute 10 tasks per cycle. For example, a node with a utilization of 50% would have an execution time of 5 cycles.

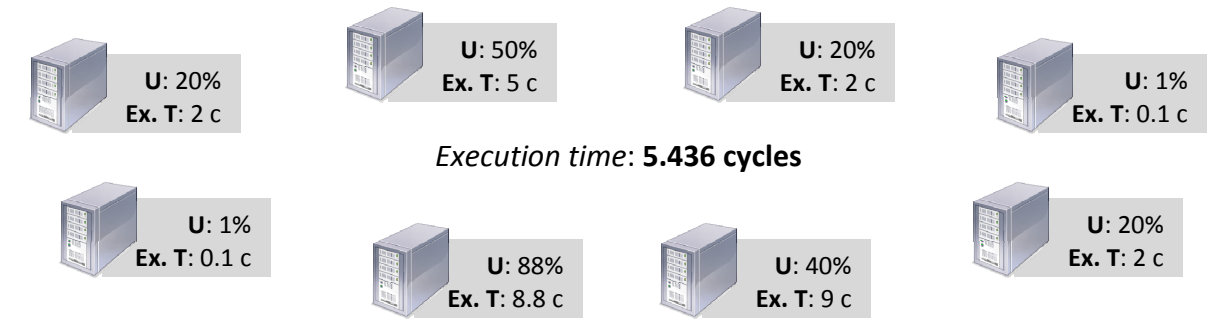


Figure 7: Execution time in the actual situation

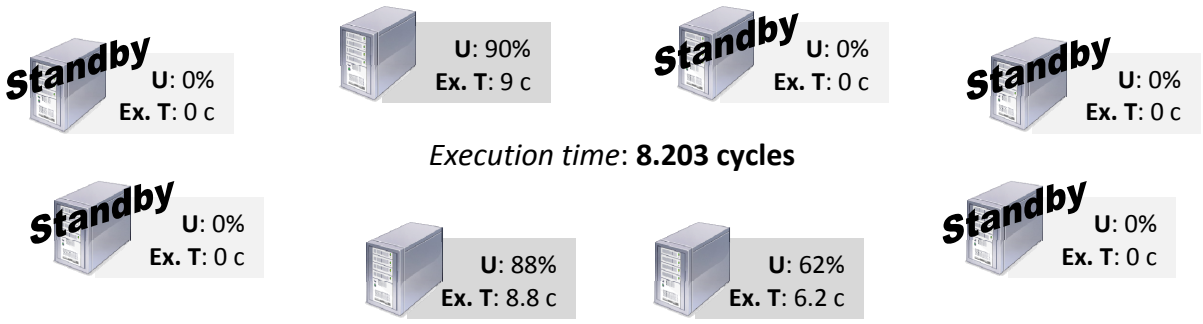


Figure 8: Execution time with energy saving

We see that, in the second case (*Figure 8*), the achievement of energy saving entails an increase of the execution time in the system.

If our only goal was to reduce power consumption we should not care about this result. However, improving the system means taking care about all the parameters. We cannot just focus on saving energy without caring on how slow the system gets to achieve that goal.

To reduce this increase of execution time when trying to reduce energy consumption we should try to distribute the load among the servers. We should not distribute the entire load among all the nodes, since this would take us back to the situation we tried to solve first (*2.3*). What we should do is balance the load according to a threshold set by the administrator of the system.

Again, we will use the example started in *2.3* to explain how load balancing should work.

After applying the rules defined for energy saving, we have the system showed in *Figure 8*.

If we set a threshold of 60% all the servers which utilization is over that value would try to distribute their load among the rest of the nodes.

In *Figure 9* we mark as overloaded (darker background) all the nodes with utilization over the set threshold (60%), and we indicate with arrows how that load can be distributed by waking up the minimum number of standby nodes.

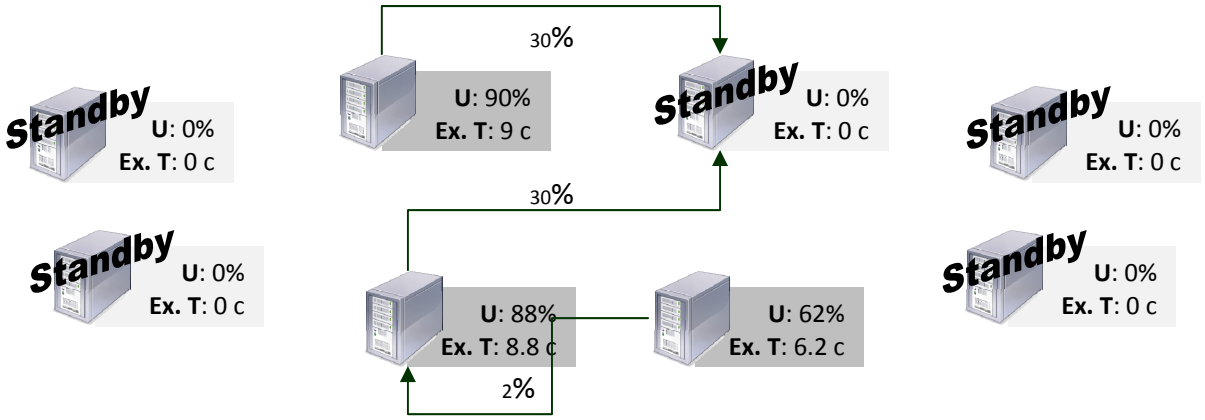


Figure 9: Proposed load balancing

After the load balancing, we see in *Figure 10* that the execution time has decreased.

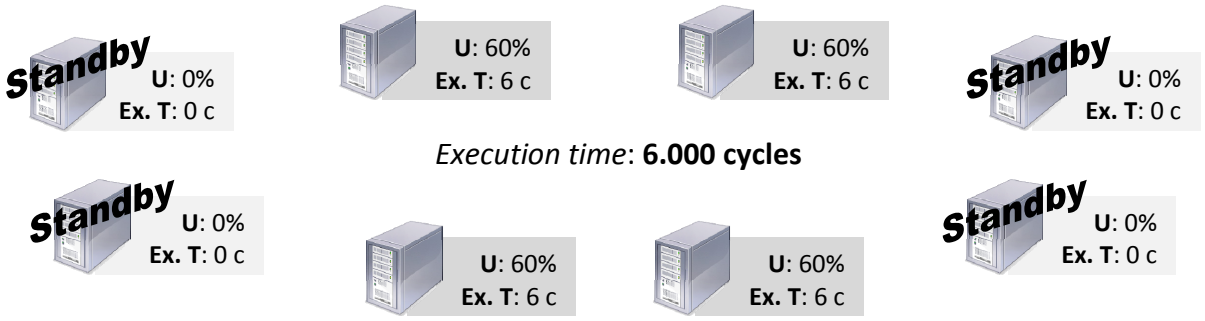


Figure 10: Execution time after load balancing

Finally, we can see in *Figure 11* that the power consumed after load balancing has increased a little comparing to the one obtained when we just cared about reducing consumption, but now we can say that we have an equilibrated situation in which we improved the performance of the system obtaining good power saving values with a reasonable execution time.

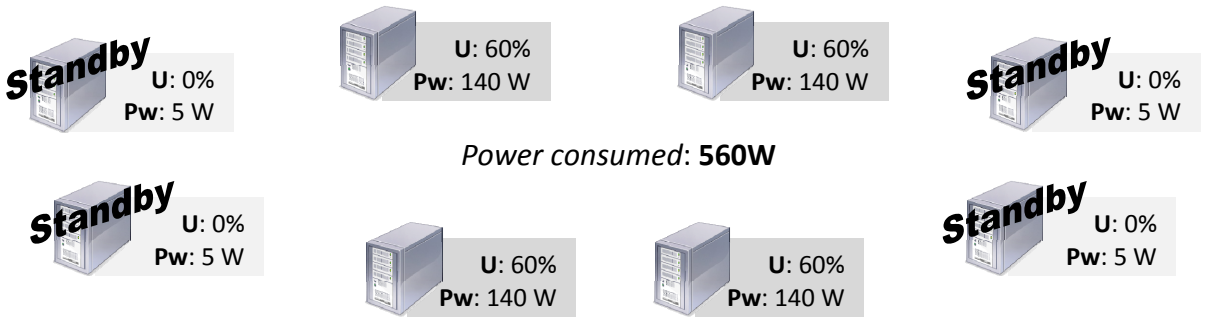


Figure 11: Power consumed after load balancing

Once again, self-organization principles are present in the decisions of how to distribute the load and which servers must wake up.

3.4 State of the art (other approaches)

In this section we will analyze some related work on power saving and load balancing.

There is a lot of literature on minimizing energy consumption as an important challenge in mobile networking. Another aspect, finite battery power of mobile computers represents one of the greatest limitations to the utility of portable computers. At this respect, [25] proposes an algorithm of reconfiguration of the overlay with the aim of extending the lifetime of the mobile terminals.

Furthermore, portable computers often need to perform power consuming activities, such as transmitting and receiving data by means of a random-access, wireless channel. The amount of power consumed to transfer the data on the wireless channel is negatively affected by the channel congestion level, and significantly depends on the MAC protocol adopted. There are some work illustrating the design and the performance evaluation of new mechanisms that, by controlling the accesses to the shared transmission channel of a wireless LAN, leads each station to an optimal Power Consumption level. Specifically, it is considered the Standard IEEE 802.11 and how to enhance its performance [26], or study distributed algorithms where nodes take decisions to stay awake or go to sleep on the base of local decision, see [27].

Energy consumption in data centers is an important issue, explored in depth in [28], where it is stated that the server energy consumption increases 9% per year. Besides this, it is remarked that an important part of those servers are most of the time idle, which makes it obvious that this consumption can be reduced. This motivation is the same that originated the work presented in this document.

From this starting point another type of studies on dynamic power management, which save power by shutting down idle devices, are originated. Several management algorithms have been proposed and demonstrated effective in certain applications as on hard disks of desktop and notebook computers. Dynamic power management (DPM) is an effective approach to reduce power consumption without significantly degrading performance. DPM shuts down devices when they are not being used and wakes them up when necessary. When a device is not used, it is called idle; otherwise, it is called busy. DPM algorithms observe request patterns and predict the length of idle periods [29].

Interesting results are proposed in [24] that start from the assumption that energy consumption in hosting Internet services is becoming a pressing issue as these services scale up. Internet services such as search, web-mail, online chatting and online gaming have become part of people's everyday life. Such services are expected to scale well, to guarantee performance (e.g., small latency) and to be highly available. To achieve these goals, these services are typically deployed in clusters of massive number of servers hosted in dedicated data centers. Each data center houses a large number of heterogeneous components for computing, storage and networking, together with an infrastructure to distribute power and provide cooling.

Data center energy savings can come from a number of places: on the hardware and facility side (e.g., by designing energy-efficient servers and data center infrastructures) and on the software side (e.g., through resource management). Chen et altri propose in [24] a software-based approach, consisting of two interdependent techniques: dynamic provisioning, that dynamically turns on a minimum number of servers required to satisfy application-specific quality of service; and load dispatching, that distributes current load among the running machines. This derives from the observation of real data and how internet works.

In a similar line of work, [30] presents a solution in which some power saving is achieved by applying two different controllers: one that manages performance and one that manages power (by controlling clock frequency). This solution also presents an adaptive learning of the system.

On the other side different approaches have been studied for dynamic load balancing, one of the first work was on diffusion schemes for dynamic load balancing on message passing multiprocessor networks [31].

One of the most important problems in distributed processing consists in balancing the work load among all processors. The purpose of load (work) balancing is to achieve better performances of distributed computations, by improving load allocation [32]. The load balancing has been applied to dynamic networks that are networks in which the topology may change dynamically: the main result of this study consists in proving the convergence toward the uniform load distribution of the diffusion algorithm on an arbitrary dynamic network despite communication link failures.

The algorithms studied in [32] are derived from the diffusion model and some of its variants. The first one, called GAE for Generalized Adaptive Exchange, is a new algorithm: it constraints one node to exchange its load with at most one of its neighbors. Another one, called relaxed diffusion, introduces a relaxation parameter in the diffusion algorithm. The relaxation parameter may dramatically speed up the convergence. A main result of the paper is that it presents the necessary and sufficient conditions so as to have convergence in the dynamic networks framework. In other words, working on dynamic networks like Internet where the communication links are not reliable at 100% (communication failure or low bandwidth communication), these algorithms can balance the loads of the system.

Another approach on load balancing is presented in [33] working with dynamic systems where nodes have different capabilities and limited knowledge about their neighbors and the whole system. To address this issue, the paper presents experiments with the usage of autonomic self-aggregation techniques that rewire such highly dynamic systems in groups of homogeneous nodes that are then able to balance the load among each others.

Breitgand et altri present in [34] another approach on load balancing, based on a centralized dispatcher unit that assigns the incoming load to the less occupied node of a set of servers checked. The main goal of this work is find out the optimal number of servers to be checked during assignation in order to minimize the execution time in the system. It is seen that this number depends on the arrival rate of the jobs and the efficiency ratio C (fraction between the actual query time and the average service time).

4. Technical approach.

4.1 Description of the system architecture.

4.1.1 Distributed server farm.

The system is composed by a number of nodes with a certain initial load, which can be manually or randomly assigned. According to the amount of load, we define the following four node states:

- *standby*: the node does not process any tasks, it is in energy saving mode.
- *low load*: the node can transfer its load and enter *standby* mode.
- *medium load*: the node can take further load.
- *high load*: the node cannot take further load.

All the nodes present in the system are of the same type and have the same behavior. They are connected in an overlay created by the simulator (*5.1*).

PeerSim [35] creates the overlay according to the number of links (N) we set in the *configuration* file (which will be explained in *5.1.3*). Every node establishes the indicated number of connections with random nodes (neighbors). The incoming links of a node also become links out of that node (if A establishes a link with B, automatically B has a link with A).

Because of the way the links are created, the real number of neighbors is usually twice the number indicated in said *configuration file*.

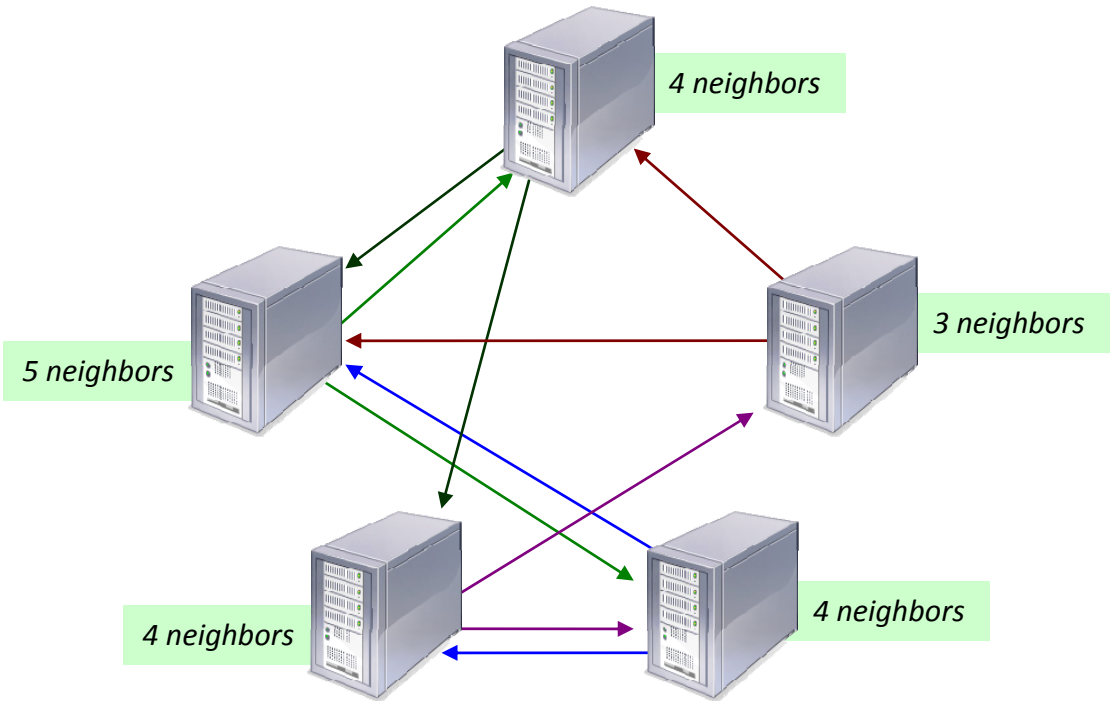


Figure 12: Example of creation of an overlay with 5 nodes and N = 2

In *Figure 12* we can see how *PeerSim* would create an overlay in a system with 5 nodes and a number of links set to 2. The arrows out of every server represent the links established with other nodes. The number beside each node is the number of neighbors it has when the creation stage is over.

As we said, each node creates two random links (because N = 2). In the end we can see that the number of neighbors of each node will be, in most of the cases, twice the number we set. In this case we can easily see that the average value is 4 (2·N).

Even though this example is really simple, we can understand that, with a more complex system, the number of neighbors will always tend to 2·N.

4.1.2 Client-Server model

Two kinds of nodes are defined in this model: *clients* and *servers*.

Servers provide a service that will be available for the clients to use.

Clients are the users in the system. They need or want to use the service provided by the *servers*. For this purpose, contracts are established between clients and servers.

In this case, all the algorithms and controls are now executed only on the servers; since it is there where we can work in real life (we cannot pretend to control the clients' computers).

For the execution of these algorithms and controls, the servers send messages to their neighbors in order to exchange the information needed during execution.

Clients are not connected in an overlay: they are connected to the servers allowing them to access the required service. These connections can change over time depending on the algorithms executed in those servers (in any case, those changes will be indicated by the servers).

Therefore, only servers are now connected in an overlay, which is created by one of the classes created for this model (*5.4.2*).

The result of the creation method is the same as in *4.1.1*, i.e.: the average number of neighbors of a server will be twice the value set as the *number of links* in the *configuration file*.

4.2 Energy saving in distributed server farm: algorithms.

Even though they are very similar, the model used in the case of constant load differs a little from that used in the case of variable load. For this reason it is necessary to divide this section in two parts. On the first one we will explain the algorithms used in our system in the case of constant load (*4.2.1*) and then, on the second part, we will talk about the modifications added for the case of variable load (*4.2.2*).

4.2.1 Constant load

4.2.1.1 Standby Algorithm

It is the algorithm responsible of achieving the energy saving.

It can be executed by any node and by more than one node at the same time. It starts at a random time, when a node starts acting as a *matchmaker*. When that happens, the *matchmaker* chooses two random neighbors in the overlay and checks their state. If one of the neighbors is in *low load* state and the other one is either in *low* or *medium load* state the node realizes that some energy can be saved. For that purpose, the matchmaker will order the *low load* node (if they are both in *low load* state it does not matter which one we choose) to transfer its load to the other neighbor. In this way, the node that sent its load goes *standby*, thus saving some energy.

This algorithm is enough to reduce the power consumption, but it does not take into account the execution time. While we get astonishing results regarding the energy saving, the execution time rockets, making the system slow. Thus, another algorithm is necessary to keep reasonable execution time values.

4.2.1.2 Overload Algorithm

In this second algorithm, each node checks periodically its execution time, which is calculated as the relationship between the number of tasks and the process capability. When this value is over a certain threshold, the node looks for a *standby* neighbor that can help it lighten its load. The procedure is very similar to the one in the *Standby Algorithm*, but now the node just selects one neighbor. When the overload node finds a *standby* neighbor, it transfers half of its load to the neighbor, which *wakes up* and enters the state corresponding to the load received.

By using the *Overload Algorithm* we reduce power saving, as it wakes *standby* nodes, but it is completely necessary if we do not want to have an extremely slow system.

4.2.1.3 Algorithms behavior

Table 1 shows the initial states for nodes A and B and those after the interaction, A' and B'.

In the *Standby Algorithm*, as we have explained, there is a third node, the matchmaker, which compares the load of nodes A and B and decides which one and when has to transfer its load.

In the *Overload Algorithm* the overloaded node (A) contacts directly a standby neighbor (B), without the intervention of a matchmaker.

Standby			
A	B	A'	B'
LL	LL	ML	SB
ML	LL	ML / HL	SB
LL	ML	SB	ML / HL

Overload			
A	B	A'	B'
HL	SB	ML	ML

Table 1: Interaction of nodes in each algorithm
(SB = Standby, LL = Low load, ML = Medium load, HL = high load)

4.2.2 Variable load

4.2.2.1 Standby Algorithm

Quite similar to the *Standby Algorithm* described in *4.2.1.1* but with a very big difference: now there is no matchmaker. Considering the results obtained in *6.1.7* we decided to remove the matchmaker, since it makes no difference and increases power consumption.

This way a node with low or medium load will get a random neighbor and check its state. If it is low load, the node executing the algorithm will ask the neighbor to send its load and go to sleep.

4.2.2.2 Overload Algorithm

This algorithm works in a similar way to the *Overload Algorithm* described in *4.2.1.2*.

When a node is overloaded it looks for a neighbor that is low loaded so it can transfer part of its load. If the node cannot find any low loaded neighbor, a standby node is awoken to take part of the overloaded node's load.

This algorithm is slightly different than the previous one, where the overloaded node always woke up a standby node to help it. By checking the low loaded nodes first we avoid waking up more nodes than necessary, which helps us maintain good saving values.

In this model, a node is considered overloaded when it satisfies the following two conditions:
- Its number of connections is over a certain value
- The difference between the login and the logout rates is positive.

A negative difference between the login and the logout rates means the load of the node is reducing. This could possibly make the node enter the normal range of load. This way we avoid unnecessary exchanges of load that could lead to an increment of the power consumed.

4.2.2.3 Underload Algorithm

This algorithm adds a behavior we did not have in *4.2.1*, opposite to the *Overload Algorithm*. Its aim is to save some energy by putting to sleep underloaded nodes.

When a node is underloaded it looks for a low or medium loaded neighbor. If that neighbor is found, it will take the load of the underloaded node so it can go standby, thus saving some power.

A node is considered underloaded when it satisfies the following two conditions:
- Its number of connections is below a certain value.
- The difference between the login and the logout rates is negative.

The same reasoning used in the *Overload Algorithm* can be applied to explain the second reason. If the difference between the login and the logout rates is positive, the load of the node is increasing. It would not have sense to transfer its load to a neighbor and go standby because then that neighbor would probably soon become overloaded.

4.3 Load balancing in client-server model: algorithms.

In this model, the states of the nodes are determined according to two new thresholds:
- *Available_threshold*: number of connections under which a node is considered free to help its neighbors.
- *Overload_threshold*: number of connections over which a node is considered to be overloaded.

The definition of the standby state remains the same as before (a node with no connections). A node with a number of connections smaller than the *available_threshold* is now considered low loaded. On the contrary, the nodes with a load over the *overload_threshold* are considered high loaded. The rest of the nodes are considered to be medium loaded.

The values for these two thresholds are set in the configuration file as usually.

4.3.1 Overload Algorithm

This is the algorithm responsible for load balancing.

When a server has low load it is available to help its neighbors. When this happens, a message is sent to a random neighbor notifying the availability of the node. If the node receiving the availability message is high loaded, the *Overload Algorithm* starts.

As shown in *Figure 13*, the sequence of exchanged messages is:

1. Node A sends an availability message to node B.
2. Node B returns to node A the list of clients that should transfer to another node so node B stops being overloaded.
3. Node A checks how much of that load it can take and notifies node B.
4. Node B tells the nodes indicated by node A to reconnect to that server (e.g., when they perform the following requests).
5. Old connections are removed from node B and new ones are created between said clients and node A.

Besides, if a node in high load state does not receive an availability notification for five cycles it looks for a standby neighbor. If it is found, the high loaded node tells some clients (those it needs to transfer to stop being overloaded) to reconnect to the standby server.

As we said, *Figure 13* shows the interactions that take place in this algorithm. Note that just the links between the clients and Server B are shown, since those initially established with Server A are not involved in the algorithm.

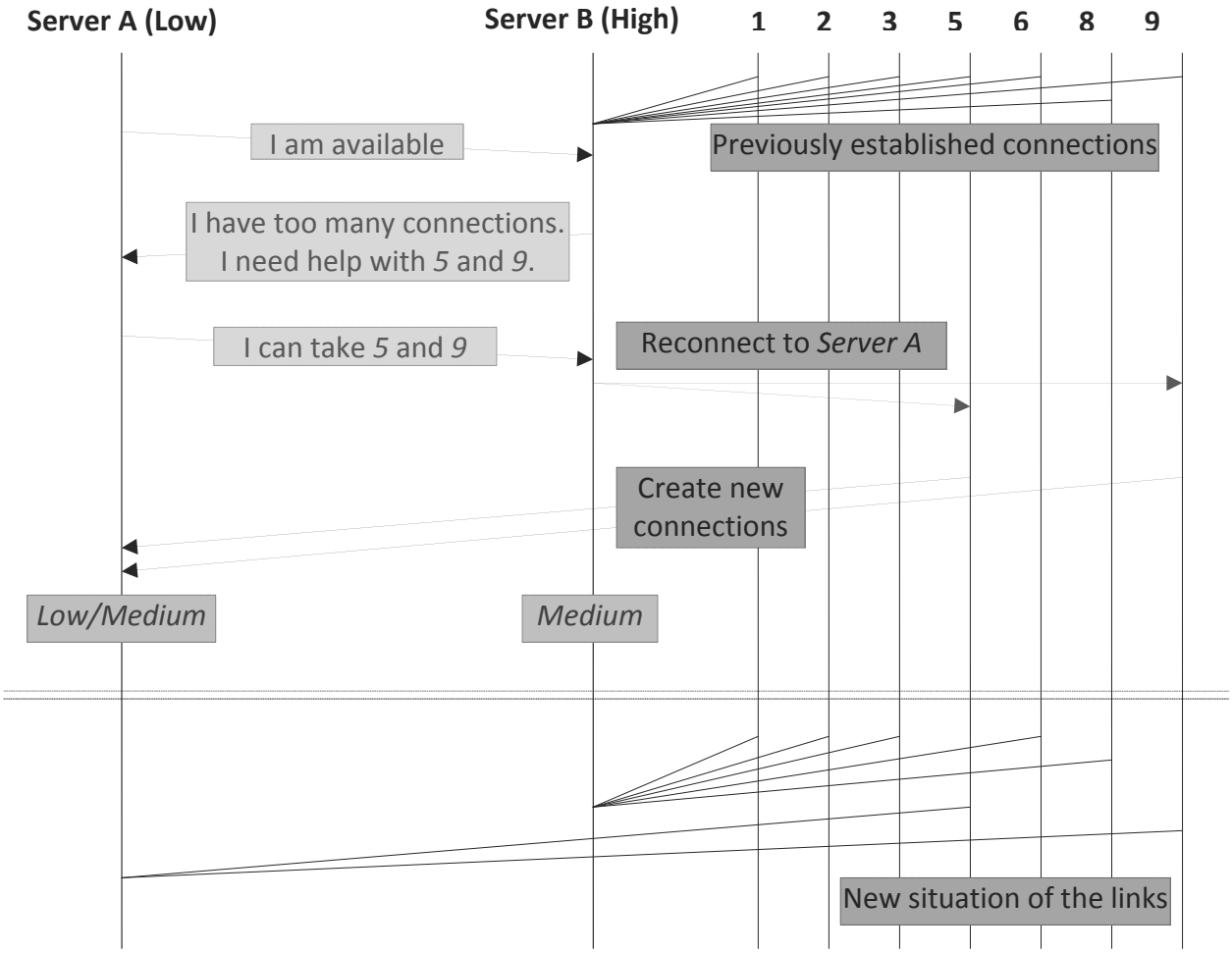


Figure 13: *Overload Algorithm*

In *Table 2* we can see the different interactions between nodes that can occur in the algorithm, being node B the node sending the availability message and node A the receiver of that notification. Since the system still considers the four mentioned states (standby, low load, medium load and high load), we use arrows (↑ and ↓) in the table to indicate that, being in a certain state, a node is close to the upper or lower one. The exchange of messages will take place as long as the states are those indicated in columns A and B, but the transference of load is determined by the amount of load in the node (indicated by the arrows).

Overload Algorithm			
A	B	A'	B'
HL	LL	ML	ML / LL
HL	SB	ML	LL

Table 2: Interaction of nodes in the *Overload Algorithm*

4.4 Load balancing and standby in client-server model: algorithms.

Besides the *Overload Algorithm* explained in *4.3.1*, we introduce a *Standby Algorithm* to achieve the desired energy saving.

4.4.1 Standby Algorithm

As happened before, when a server has low load it is available to help its neighbors, so a message is sent to a random neighbor notifying this availability. If the node receiving the availability message is low or medium loaded, the *Standby Algorithm* starts.

As we can see in *Figure 14*, this algorithm works as follows:

1. Node A sends an availability message to node B.
2. Node B returns to node A the list of clients with which it has a link.
3. Node A determines how much of the neighbor's load it can take and sends a message with this information to node B.
4. Node B tells the nodes indicated by node A to reconnect to that server.
5. Old connections are removed from node B and new ones are created between said clients and node A.

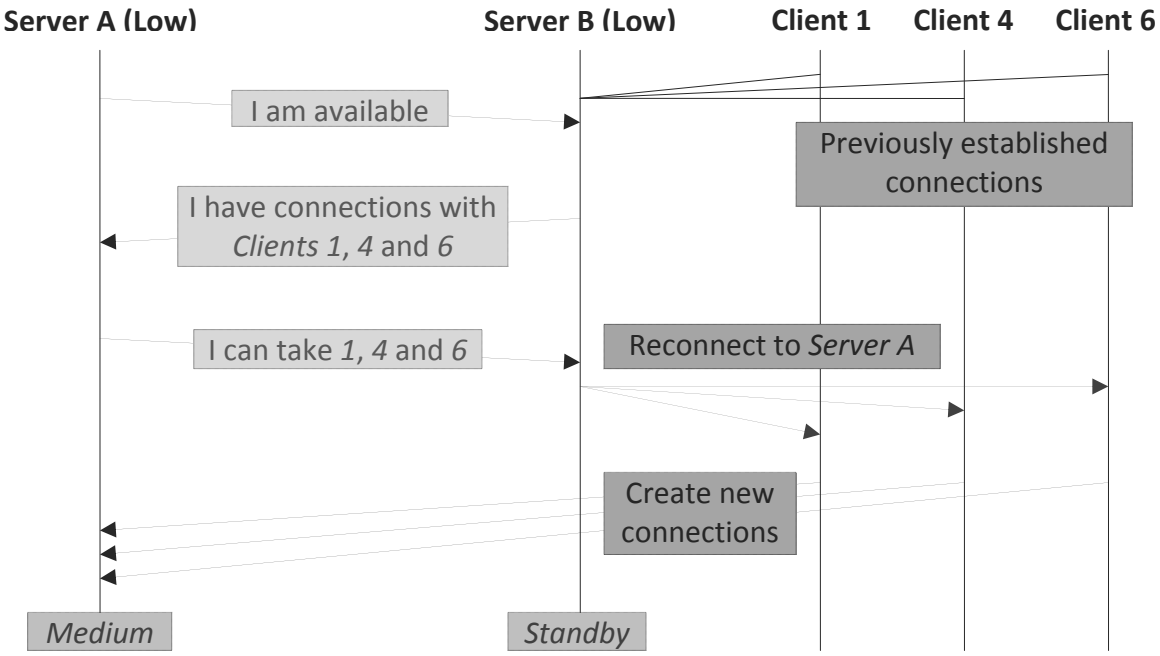


Figure 14: *Standby Algorithm*

In *Table 3* we can see the different interactions between nodes that can occur in the algorithm, being node B the node sending the availability message and node A the receiver of that notification.

Standby Algorithm			
A	B	A'	B'
LL	LL	ML / LL	SB
ML	LL	LL / ML	ML

Table 3: Interaction of the nodes in the *Standby Algorithm*

5. Simulation tool

5.1 PeerSim

5.1.1 Description

Peer-to-peer (P2P) systems can be extremely large scale (millions of nodes). Nodes in the network join and leave continuously. Experimenting with a protocol in such an environment is no easy task at all.

PeerSim has been developed by the *University of Bologna* to cope with these properties and thus to reach extreme scalability and to support dynamism. In addition, the simulator structure is based on components and makes it easy to quickly prototype a protocol, combining different pluggable building blocks, which are in fact Java objects [35].

PeerSim 1.0 supports two simulation models: the *cycle-based* model and a more traditional *event-based* model. In this document we use the former.

The *cycle-based* model is a simplified one, which makes it possible to achieve extreme scalability and performance, at the cost of some loss of realism. Several simple protocols can tolerate this loss without problems, according to our own experience; yet, care should be taken when this model is selected to experiment with a protocol.

The simplifying assumptions of the cycle based model are the lack of transport layer simulation and the lack of concurrency. In other words, nodes communicate with each other directly, and the nodes are given the control periodically, in some sequential order, when they can perform arbitrary actions, such as call methods of other objects and perform some computations.

5.1.2 Simulation life-cycle

PeerSim was designed to encourage modular programming based on objects (building blocks). Every block is easily replaceable by another component implementing the same interface (i.e., the same functionality). The general steps we should take for a simulation are:

1. Choosing a network size (number of nodes).
2. Choosing one or more protocols to experiment with and initializing them.
3. Choosing one or more *control* objects to monitor the properties we are interested in and to modify some parameters during the simulation (e.g., the size of the network, the internal state of the protocols, etc).
4. Running the simulation invoking the *Simulator* class with a configuration file that contains the above information.

All the objects created during the simulation are instances of classes that implement one or more interfaces. The main interfaces to become familiar with are:

- *Node*: the P2P network is composed of nodes. A node is a container of protocols. The node interface provides access to the protocols it holds, and to a fixed ID of the node.
- *CDProtocol*: it is a specific protocol designed to run in the cycle-driven model. Such a protocol simply defines an operation to be performed at each cycle.

- *Linkable*: typically implemented by protocols, this interface provides a service to other protocols to access a set of neighbor nodes. The instances of the same linkable protocol class over the nodes define an overlay network.
- *Control*: classes implementing this interface can be scheduled for execution at certain points during the simulation. These classes typically observe or modify the simulation.

The life-cycle of a cycle-based simulation is as follows.

The first step is to read the configuration file, given as a command-line parameter (see *5.1.3*). The configuration contains all the simulation parameters concerning all the objects involved in the experiment.

Then the simulator sets up the network initializing the nodes and the protocols in them. Each node has the same kinds of protocols; that is, instances of protocols form an array in the network, with one instance in each node. The instances of the nodes and the protocols are created by cloning. That is, only one instance is constructed using the constructor of the object, which serve as prototype, and all the nodes in the network are cloned from this prototype. For this reason, it is very important to pay attention to the implementation of the cloning method of the protocols.

At this point, initialization needs to be performed to set up the initial states of each protocol. The initialization phase is carried out by *control* objects that are scheduled to run only at the beginning of each experiment. In the configuration file, the initialization components are easily recognizable by the *init* prefix. Please note that we talk about *initializer* objects just to remark their function and to distinguish them from ordinary *control* objects, but the initializer objects are simply controls configured to run in the initialization phase.

After initialization, the cycle driven engine calls all components (protocols and controls) once in each cycle, until a given number of cycles or until a component decides to end the simulation. Each object in *PeerSim* (controls and protocols) is assigned a *scheduler* object which defines when they are executed exactly.

By default, all objects are executed in each cycle. However, it is possible to configure a protocol or control to run only in certain cycles, and it is also possible to control the order of the running of the components within each cycle. The latter scenario is illustrated in *Figure 15*.

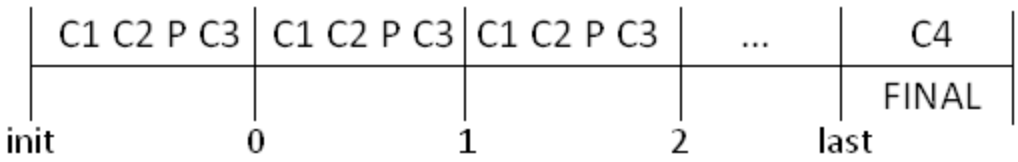


Figure 15: Scheduling controls and protocols

'C' letters indicate a control component, while letter 'P' indicates a protocol. The numbers in lower part of the picture indicate the *PeerSim* cycles. After last cycle, it is possible to run a final control to retrieve a final snapshot.

When a *control* has to collect data, they are formatted and sent to standard output and can be easily redirected to a file to be collected for further work.

5.1.3 Configuration file

It is the most important file for a *PeerSim* simulation, because it defines the functions that will be used as well as their parameters. This file gathers all the configurable parameters, so it is very simple to set different simulation scenarios just by changing it.

We can distinguish between the generic *PeerSim* parameters and those specific for our application. In the first group we find the following:

- *Size of the network*: number of nodes in the system.
- *Number of links*: number of neighbors each node has.
- *Number of cycles* in the simulation.
- *Number of experiments*: number of simulations that will run with the same parameters.

Those on the second group will be explained later, since they are not the same in the different models we created.

Besides, there are some others that do not need to take a value: they will be considered only if they are included in the configuration file. We can call them *boolean parameters*.

Towards the end of the file we find a line in which we can decide which controllers will be executed in the simulation. This line starts with the word *include*.

The decision of the controllers that will be executed is made by writing the short name given to each one of them previously on the configuration file.

The configuration file is specified on the command line as follows:

java peersim.Simulator config-file.txt

5.2 Classes for energy saving (constant load)

5.2.1 LoadDistribution.java

Before the nodes can start running their tasks, it is essential that their load is initialized. We cannot do this in the protocol (*MatchMaker.java*) since the simulator creates it once and then clones it to each one of the nodes, and we may need different initial values. For this reason, the definition of an initializer class (as we explained in *5.1.2*, a controller that will be executed just in the beginning) is necessary. In our simulation, this class is called *LoadDistribution.java*.

Its constructor is responsible for taking the parameters defined in the configuration file. Among these parameters we find *low*, *medium* and *high*, which represent the percentage of the number of nodes that will be in every state. If the variable *random* is set, these values will be generated randomly.

The initialization of the nodes takes place in the main method, called *execute()*, which is just executed once in the beginning of the simulation. It first calculates the *initial_time* (average execution time in the system) and then, according to the values got in the constructor, sets the load of every node, calls the *adjustPower()* method to set also the power and state values, and sets the *initial_time* in every node.

The load is distributed as follows:
- *low* nodes get a random value in the interval [23-33].
- *medium* nodes get a random value in the interval [56-66].
- *high* nodes get a random value in the interval [89-99].
- if there are any remaining nodes (in the configuration file the sum of the nodes specified in every state is under 100%), they are initialized with a load of 0 (this does not happen with the random initialization).

5.2.2 MatchMaker.java

MatchMaker.java is the main class. It represents the functionalities offered by each of the nodes in the system. The constructor method of this class initializes its variables and gets the parameter values defined in the configuration file, which will be described in the next chapter of this paper.

For making the simulation faster, if there are not available nodes for the algorithms to work (standby nodes in the *Overload Algorithm* and low loaded ones in the *Standby Algorithm*) they are not executed. These situations are notified to the nodes through two variables. In the real implementation there would not be any global variable controlling this, since we are aiming for a decentralized model.

This class includes some methods that will be accessed from the main method of this class, that will be explained later, or from another classes such as the controllers. These methods are:
- *setInitialTime(double initial_time)*: sets the initial time, i.e., the average execution time in the system at the beginning of the simulation.
- *setNoLowLoad(boolean val)*: sets the variable that tells the node if there are still low loaded nodes or not.
- *setNoStandby(boolean val)*: sets the variable that tells the node if there are still *standby* nodes or not.
- *int getPower()*: returns the power consumption of the node according to *Table 4*.
- *int getState()*: returns a numerical value representing the state of the node (0 for *standby*, 1 for *low-load*, 2 for *medium-load* and 3 for *high-load*).
- *modifyValue(double var)*: adds (if positive) or subtracts (if negative) the given quantity to the value representing the load in the node.
- *checkOverload()*: calculates the execution time of the node and compares it to the initial_time. If it is over the threshold defined in the configuration file, it marks the node as overloaded. It also removes this mark when the node has reached a normal execution time value.
- *adjustPower()*: sets the power consumption and state of the node according to its load. The values are assigned as shown in *Table 4*.

Load	State	Power consumption
0	0 (standby)	5
>0 and ≤33	1 (low load)	125
>33 and ≤66	2 (medium load)	140
>66	3 (high load)	155

Table 4: State and power consumption assigned depending on the load

There are three additional functions that should be included when we want to execute an average execution time control. This can be used just to get an idea of an ideal situation, but it could not be implemented in a real system, since it is very difficult to have a global control of the whole overlay.

- *setOver(boolean over)*: sets the variable that tells the node that the whole system is overloaded.
- *setOverloaded(boolean over)*: sets the variable that marks the node as overloaded.
- *boolean isOverloaded()*: returns true if the node is overloaded.

The main method in *MatchMaker.java* is called *nextCycle* (required by *PeerSim*). The code included in this method is the one that will be executed in the node each cycle of the simulation. The implementation of the two algorithms explained in the previous chapter takes place here.

First of all, if the node is overloaded and there are still nodes in *standby* in the system, the *Overload Algorithm* is executed. The node gets a random neighbor and checks its state. If it is in *standby*, the node wakes the neighbor and transfers half of its load to it. *adjustPower()* is then called in the node and its neighbor, to set the right state and energy values according to the new load.

After this, the *Standby Algorithm* starts if there are still some low loaded nodes (otherwise this step is skipped). A random number is generated and compared to the *threshold of activation* set in the configuration file. This determines if the node will act as a matchmaker or not in this cycle. When it does, a variable called *match* is set to true, so the same node cannot be a matchmaker in the following cycles according to a value set in the configuration file (5 cycles by default).

The matchmaker gets two random neighbors and compares their states.

One possibility is that the two neighbors form a couple *low-load* / *medium-load*. In that case, the medium loaded neighbor transfers its load to the low loaded one. *adjustPower()* is called then to update the state and power values.

Another possibility is that the two of them are in *low-load* state. If that happens, the node proceeds as in the first case, taking the first of the neighbors as the medium loaded.

If the node finds the neighbors in any other combination of states, no load exchange takes place.

5.2.3 Controllers

To see the results of our simulations on the screen and to be able to save them in files, we define the next controller classes: *LoadObserver.java*, *TimeReportObserver.java* and *ExecutionTimeObserver.java*.

LoadObserver.java is responsible for the control of energy saving and node state. Besides the main method *execute()*, that runs in each cycle, it has the function *varLoad()*, which is invoked in *execute()*. *varLoad()* simulates the variation of the load in a node to make it more realistic. The variations consist on random values that are assigned to the nodes by calling the *MatchMaker.java* method *modifyValue(int var)*, followed by *adjustPower()*.

In the method *execute()*, this controller has different parts, which will be executed depending on which variables are set on the configuration file:

- If *show_load* is set the load of every node will be shown on the screen while the simulation is running. This option can be useful when working with small networks, but it is useless when working with a large network.
- If *show_node_power* is set, we will see an array similar as the one that we get by setting *show_load* but with power instead of load values.
- If *generate_report* is set, a text file containing the values of power saving will be saved. It will be named *Saving_Report_XXXXX.txt*, where XXXXX is a random number between 0 and 100,000.
- If *save_state_report* is set, the evolution of the number of nodes in every state will be saved in a text file named *State_report_XXXXX.txt*, where XXXXX is the same random number as in the saving report.

These text reports will be used to generate the *Excel* charts that will be presented later in this document.

The first action taking by this method, after resetting all the variables, is calling *varLoad()*, which increments/decrements the load of each node between 0 and 4 unities. After calling this method, the *LoadObserver* enters a loop in which some actions are performed with every single node, so the control of the system can be made. If, according to the configuration parameters, it is going to be necessary, the observer gets the node power and the sum of all the nodes, saving this as the actual power. A control of the number of nodes in each state will also be made in this loop. Besides, the *MatchMaker* method *checkOverload()* will be invoked for every node, so we are certain that the overload control is done after the load-exchange algorithms.

When the loop ends, we check if the system ran out of standby or low loaded nodes. When any of those situations occur, all the nodes in the system get notified by a call to the already mentioned methods *setNoStandby(boolean val)* and *setNoLowLoad(boolean val)*. If the situation changes in the next cycles, the nodes get notified with the same methods. This functionality in included to make the simulation run faster (if there are no more *standby* nodes the *Overload Algorithm* will not make any changes, and the same happens with the *Standby Algorithm* when there are no more *low-load* nodes), but it would not be implemented in the real system, since it is impossible to have a global control of all the nodes in the network.

Finally, and only if required, the energy saving is calculated as the relationship between the power difference (initial power – actual power) and the initial power.

TimeReportObserver.java generates a file with the average execution time in the system at the end of each cycle. It is necessary to indicate, as a parameter in the configuration file, the process capability of the nodes. This represents the number of tasks that one node is able to run in one cycle, and it is used to calculate the execution time based on the load of each node.

Optionally, we can tell the observer to show the average execution time on the screen every cycle by setting the parameter *show_time*.

ExecutionTimeObserver.java allows us to have a global control of the system, but it is not a realistic situation, so it should be used just for having an idea of what the best situation would be.

If this controller is included, the decision of when to start the *Overload Algorithm* is not taken locally. This class, in its method *execute()*, gets the execution time of every node in the system and calculates its average. When it is over a certain threshold set in the configuration file, it tells all the nodes that the system is overloaded (by calling the *MatchMaker* method *setOver()*) and marks the nodes with an execution time over the threshold as overloaded (by calling the

MatchMaker method *setOverloaded()*). When the average time returns to the time defined as *normal*, all the nodes are notified so the *Overload Algorithm* stops executing.

5.2.4 Configuration file

Besides the *PeerSim* parameters listed in *5.1.3*, this model has the following parameters, used by the classes we created:

- *Process capability* of the nodes.
- *Matchmaker threshold*: probability for a node to act as a matchmaker. Must be a value between 0% and 100%.
- *Matchmaker limit*: when a node acts as a matchmaker, it has to wait this number of cycles until it can be activated as a matchmaker again. Must be a natural number.
- *Slow time*: when the execution time of the node is over this value, it is considered to be overloaded. It represents percentage over the initial execution time.
- *Normal time*: when the execution time of the overloaded node goes below this value, it is considered to be running normally again. It represents percentage over the initial execution time.
- Load parameters *low*, *medium* and *high* represent the amount of nodes initially in every state. These values must be between 0% and 100%, and the sum of the three of them cannot be greater than 100. If the initial load is set to random, these parameters are not taken into account.

The following functionalities can be selected with the *boolean parameters*:

- Show which nodes act as a matchmaker and which ones exchange their load (*protocol.mmk.show_match_maker*).
- Show the load of all the nodes (*control.obs.show_load*).
- Show the power of all the nodes (*control.obs.show_node_power*).
- Show the global energy saving, compared to the initial power consumption (*control.obs.show_power_saving*).
- Save a file containing the energy saving values (*control.obs.generate_report*).
- Save four files with the number of nodes in every state (*control.obs.save_state_report*).
- Show the average execution time (*control.tro.show_time*).
- Randomize the distribution of the initial load (*init.load.random*).

This is the list of the protocols in this model, with their short name in parentheses, to use it in the *include* line.

- *LoadObserver* (*obs*): energy saving and load observer.
- *TimeReportObserver* (*tro*): local management of overload situations.
- *ExecutionTimeObserver* (*ext*): global management of overload situations.

While *LoadObserver* should always be included, we can switch between the second and the third, depending on which kind of overload control we want for the system. We can also exclude both of them if we do not want to monitor the execution time. For local decision, the *include* line will look like this:

include.control obs tro

In case we want global control, we should simply write *ext* instead of *tro*.

Finally, for the generation of a file containing the average execution time we have two different options, depending on which kind of overload control we are using. When overload is managed locally, to save this file is enough with the inclusion of the *TimeReportObserver* in the *include* line. In a global control situation, the next line has to be included in the configuration file:

control.ext.generate_report

5.3 Classes for energy saving (variable load)

We modify the classes created in *5.2*, introducing the changes needed in this model. The main changes are the way traffic is modeled, which now is more realistic; and that we do not use the matchmaker anymore: the nodes contact directly their neighbors.

The variations introduced are related to the studies about utilization of the nodes and traffic model included in [24].

We changed the way the load and the energy consumed by the nodes are modeled. In the previous system, each of the nodes had a value, between 0 and 100, representing the tasks it held, i.e., its load. Depending on this value, the state of the node was determined (standby, low, medium or high) and a fixed power consumption value was assigned to each state.

The revised model adds new variables, making it closer to reality. These variables are:

- *Number of connections* (N): represents the number of active connections in server, i.e. the number of clients currently connected to the node.
- *Login rate* (LI): in [24], it represents the number of incoming connections per second in a node; but in our model is the number of incoming connections per minute, since every simulation cycle equals to a minute.
- *Logout rate* (LO): represents the number of disconnections per minute.

With this we can calculate the CPU utilization (U) with the following formula, given in [24]:

$$U = 2.84\ x\ 10^{-4} \cdot N + 0.549 \cdot LI - 0.820$$

The energy consumed (in Watts) by a node will be calculated dynamically from this value with the following formula:

$$Power = \begin{cases} 110 + \dfrac{(15 \cdot U)}{33} & if\ state\ \neq 0 \\ 5 & if\ state = 0 \end{cases}$$

The figure of the matchmaker has been eliminated, since it did not make any important difference (this can be seen in *6.1.7*). Now each node interacts directly with a neighbor, reducing to two the number of nodes involved in the process.

The way the load is transferred has changed too. In the previous model we assumed that one node sent its tasks to the other, so the number of tasks in the receiving node was the sum of its own tasks and those of its neighbor.

Now, when we talk about load transference we mean that the receiving node will get just the login rate of the neighbor (or a part of it) and add it to its own LI. This way the number of connections is not transferred from one node to the other, since it is not realistic that an already

established connection is transferred without logging out from the actual node and logging in to the new one.

In the previous model the load was constant through all the simulation. In this new approach, we take into account the observations about the variation of the number of connections and the login rate included in [24]. We use them as a pattern for the variation of our login and logout parameters. This way we get a more realistic model, where LI and LO are different depending on the time of the day.

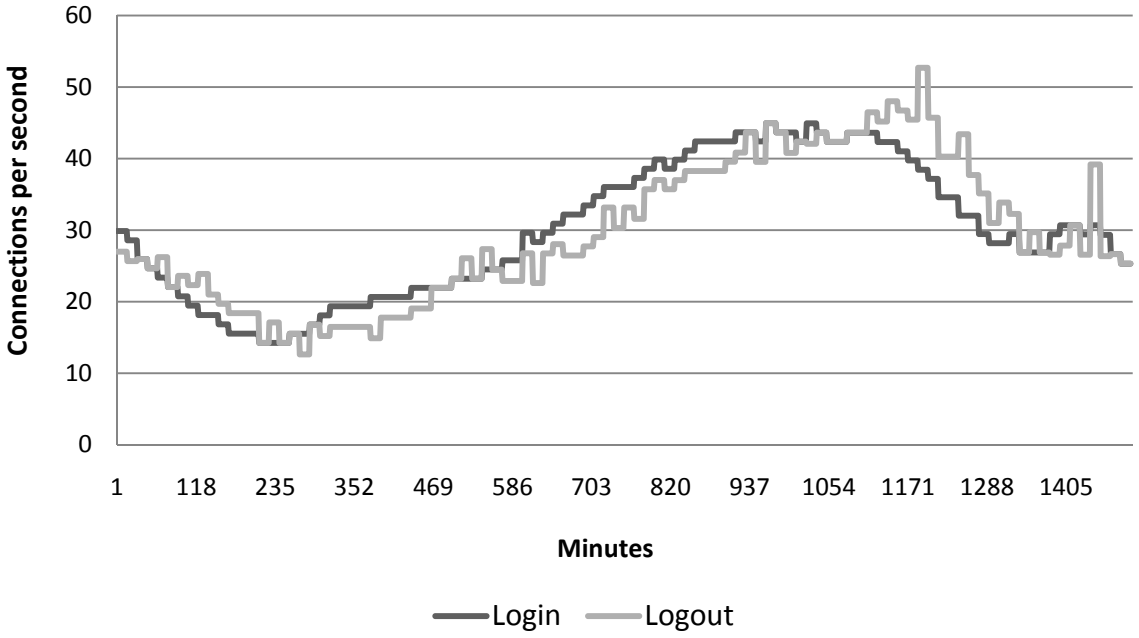


Figure 16: Average login and logout rates throughout a day

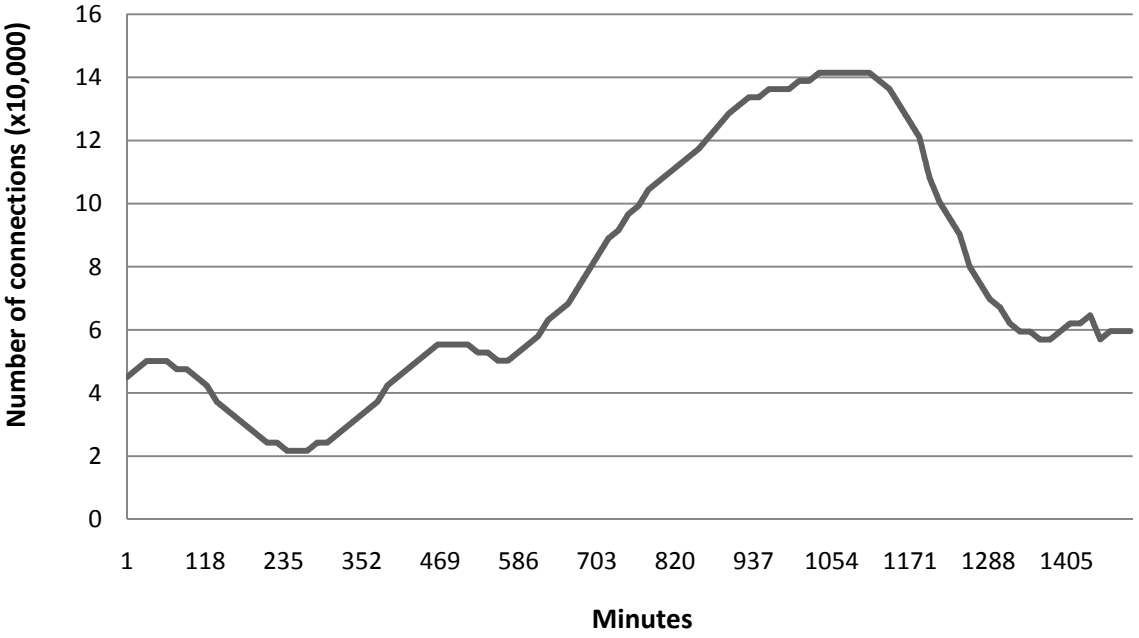


Figure 17: Average number of connections throughout a day

The figures above show the average login/logout rates that we use in this revised model (*Figure 16*) and the resulting number of connections (*Figure 17*). The results shown represent the variation throughout a day, since the simulation time was 1,500 cycles and each cycle stands for a minute (1 day = 1,440 minutes).

5.3.1 LoadDistribution.java

The only difference with *5.2.1* is the values that are now assigned depending on the states:
- *Low load*: N between 40,000 and 50,000; LI between 1,500 and 2,100 (25-35 connections per second). With these values, the initial CPU utilization is between 24.27% and 32.60%.
- *Medium load*: N between 50,000 and 60,000; LI between 4,800 and 5,400 (80-90 connections per second). With these values, the initial CPU utilization is between 57.30% and 65.63%.
- *High load*: N between 70,000 and 80,000; LI between 6,000 and 6,600 (100-110 connections per second). With these values, the initial CPU utilization is between 73.96% and 82.29%.

5.3.2 MatchMaker.java

This class is again the core of our system. The algorithms described take place here.

Most of the methods included in *5.2.2* were deleted because they were no longer necessary. Just the methods *getPower()*, *getState()* and *adjustPower()* remain. While *getPower()* and *getState()* are exactly the same, *adjustPower()* now calculates the utilization of the node and its power with the formulas already shown and sets the state depending on the U.

Besides these methods and the main one, some new methods are defined:
- *setN(int NumberOfConnections)*: sets N to the given value.
- *int getN()*: returns the value of N.
- *setL(int LoginRate)*: sets LI to the given value.
- *int getL()*: returns the value of N.
- *setLO(int LogoutRate)*: sets LO to the given value.
- *int getLO()*: returns the value of LO.
- *variation()*: responsible of updating the number of connections in the node and checking if it is overloaded or the number of connections is very small. This method can act in two different ways, depending on the value of the login rate.

  If LI equals 0 it means that the node is either shutting down or already in standby. The number of connections is checked then to know in which of those two situations the node is. If the value of N is 0 the node is in standby and no further action has to be taken. Otherwise, N is reduced by two times the logout rate (the departure rate in a node is higher when it is shutting down). If after this reduction N is 0 or less, N is set to 0, the state to standby (by setting it to 0 too, since it is the value assigned to this state) and the power to 5 Watts.

  For any other value of LI, N varies this way:

$$N = N + LI - LO,$$

this representing the normal variation because of the difference between the login and logout rates. After this, the number of connections is compared to the *low_limit* and *high_limit* thresholds.

If N is below *low_limit* and LI – LO is negative, the node is marked by setting the variable *below* to true. This means that the node has a very small number of connections and shutting it down would be a better option, so in the next cycle the node will try to transfer its load and go to sleep.

If N is over *high_limit* and LI – LO is positive, the node is marked as overloaded by setting the variable *over* to true. This means that the number of connections is very high and it should share some of its load with a neighbor in order to keep the system working properly.

As we saw in *5.2.2*, the main method in this class is called *nextCycle*. We focus on the things that changed from that model to the actual one, since the *limit* and *threshold* controls remain the same.

A standby node will not do anything in the *nextCycle* method until it gets awoken by an overloaded node. For the rest of the states, and depending on the situation of the node, the algorithms explained before can be executed.

First of all, if *below* is true, the node gets a random neighbor and checks its state. If it is low or medium load, the node transfers its LI to its neighbor (by using the methods *setL()* and *getL()*) and sets its own LI to 0. *adjustPower()* is called then for both nodes and *below* is set to false.

On the contrary, if *over* is true, the node selects a random neighbor and checks its state. If it is low load, the node sends half of its LI to the neighbor, and then calls *adjustPower()* for the two nodes involved in the process. After his, *over* is set to false. If the state of the neighbor is not low, the node takes another neighbor. If after five attempts the server has not found any low load node, it checks if the last neighbor taken is in standby and if so wakes it up by transferring half of its LI.

Lastly we have the main algorithm, which is an extension of the *Standby Algorithm* in *5.2.2*. Besides checking the *threshold* and the *limit*, we check also the state of the node. Since now it is the node itself who takes the neighbor's load, this algorithm will only take place if it is in low or medium load state.

After that, one random neighbor is selected and its state checked. If the neighbor is low loaded and its LI is different than 0 (if not it means that it is being shut down) the node adds the neighbor's LI to its own and calls *adjustPower()*. Finally, *setL(0)* and *adjustPower()* are then called for the neighbor.

5.3.3 Controllers

To see the results of the simulations, we keep the controller *LoadObserver.java* from *5.2.3* (with some modifications) and add a new one: *LObserver.java*.

Regarding *LoadObserver.java*, the first thing we notice is that *varLoad()* has been eliminated, since now the variation is more complex and a new controller, *LObserver.java*, has been created for that purpose.

As for what we could see on the screen, *show_node_power* remains the same and *show_load* shows now the number of connections of each node.

The *state report* is still available. The *power report* now saves the energy consumed by the overall system in a file with the name *Power_Report_XXXXX.txt*, where XXXXX is the same random number used for the state report.

The *execute()* method is almost the same as it was before, but now for each node the *MatchMaker.java* methods *variation()* and *adjustPower()* are called, so the number of connections is updated in the end of each cycle.

The new class *LObserver.java* is quite simple, since it was created for something very specific: the variation of the login and logout rates. Besides the main method, *execute()*, it has another method called *variation()*, responsible of adapting the values according to the charts included in [24]. Every 15 minutes (cycles) LI and LO must be updated. *variation()* does it, depending on what time it is, so the model is closer to reality.

If *generate_report* is true, *execute()* gets the sum of the number of connections (with *getN()*), login rates (*getL()*) and logout rates (*getLO()*) of all the nodes and calculates the averages, which are saved into files with the names *Connections_Report_XXXXX.txt*, *Login_Report_XXXXX.txt* and *Logout_Report_XXXXX.txt*, respectively, where XXXXX is a random number.

execute() also counts the number of cycles that have been executed. Every time this number is 15, *variation()* is called and the cycle count is set to 0.

5.3.4 Configuration file

Since there are a lot of things that has not changed in the configuration file from that in *5.2.4*, we will just mention the differences with respect to that section.

All the parameters with configurable values remain the same, with the exception of *process*, *slow_time* and *normal_time*, which are no longer necessary. Two new parameters are added: *low_limit* and *high_limit*. They set the thresholds used in *MatchMaker.java* to determine if the node should go standby or is overloaded.

Regarding the *boolean parameters*, *show_power_saving* disappears, since it is not possible to watch the savings on the screen anymore. Logically *show_time* is not included either, since the class in which it was used has been removed from the model. There is one new line that allows us to choose if we want to save the login/logout and number of connections reports:

control.logobs.generate_report

Besides that, the *include* line (used to choose which controllers will be executed) has been removed, since now there are only two controllers and they are necessary.

5.4 Classes for load balancing

The code is composed by three packages corresponding to the three kinds of classes *PeerSim* uses: initializers, protocols and controllers.

We have one initializer, *LoadDistribution.java*; one protocol, *Algorithms.java*; and two controllers, *Reports.java* and *LoadPeak.java*.

5.4.1 LoadDistribution.java

First of all, we assign a unique ID to each of the nodes, set their type (client or server) and create their set of neighbors by invoking the methods created for this purpose in *5.4.2* (we will explain them later).

After this, it is time to establish the initial links between clients and servers. Each client will create a randomly generated number of contracts to a random server depending on the request rate indicated in the configuration file (low, medium or high). The initial values are:

- Between 10,111 and 11,111 for *low* nodes.
- Between 21,222 and 22,222 for *medium* nodes.
- Between 32,333 and 33,333 for *high* nodes.

5.4.2 Algorithms.java

This is the main class in the simulation. It models the behavior of the nodes in the system, mainly the servers, since in this first approach to the client-server model clients do not do much.

Besides the main method (as always, *nextCycle()*), the following can be found in this class:

- *setNodeID(int id)*: sets the ID of the node to the given value.
- *setType(int type)*: sets the type of the node to the given value (0 for clients and 1 for servers).
- *setNeighbors(int links, int type0, int type1)*: when invoked in a client, it chooses a random server to which it is initially connected. In the servers it generates the overlay in the following way:
    - For each server, a list of random numbers (in the interval of the server's IDs) of size *links* is generated.
    - If any ID is repeated it is deleted from the list.
    - If a node A has B in its list, A is automatically included in the list of B.
    - Each server has finally a list of its neighbors.
- *setL(int LoginRate)*: sets the login rate of the client to the given value.
- *int getL()*: returns the login rate of the client.
- *int getN()*: returns the number of connections of the server.
- *double getPower()*: returns the power consumed by the server: 5 if it is in standby and *110 + (Number of connections)/2,000* otherwise.
- *int getState()*: returns a number representing the state of the server: 0 for standby (0 connections), 1 for low load (less than 33,334 connections), 2 for medium load (less than 66,667 connections) and 3 for high load (more than 66,666 connections).
- *setBusy(boolean busy)*: sets the variable *busy* true if the server is having a *conversation* with other neighbor and false when that *conversation* ends.
- *boolean isBusy()*: returns the state of the variable *busy*.
- *addContract(int client_id)*: when a server invokes it, a new link is established with the client identified by *client_id*.
- *int removeContracts(int num)*: tries to remove *num* contracts from the server that invokes it. If the number of contracts is lower than *num* the method removes them and returns the difference between those two values (otherwise the return value is 0). This function is used when we create a load peak in the simulation.
- *ArrayList<Neighbor> sendAvailable()*: represents the message a server sends to a neighbor when it is available to help and the answer to that message. First of all, it sets *busy* true to

avoid that a server can interact with more than one neighbor at a time. Then, if the server is high loaded and the *Overload Algorithm* is being executed it returns the list of clients that, if reconnected to another server, would make the server be in medium load.
- *getLoad(ArrayList<Neighbor> candidates, int peer_id, int protocolID)*: represents the second message the available node sends and the actions taken by the receiver. The receiver of the message tells the clients the sender will take (indicated by *candidates*) to reconnect to that node (identified by *peer_id*) and updates its number of connections.
- *reconnect(int new_contract, int protocolID)*: establishes a new link between the client and the server identified by *new_contract*.

We have also included a set of methods that will allow the nodes to calculate the average load in the system by exchanging their load information with their neighbors. We want the average to be calculated at the end of each cycle, when the servers are done with the algorithms. For this reason we will have to call these methods in a controller class.

In the current implementation of the model we do not use this functionality, but in the future the decisions taken by the nodes could rely on this value.
- *resetAverage()*: resets the values used to calculate the average.
- *sendLoadInfo(int pid)*: the server sends its load information to all its neighbors by calling the following method.
- *sendNumberOfConnections(int node)*: simulates the message in which a node gets the load information from a neighbor. The server then stores the value as well as the number of load messages received until that moment.
- *double calculateAverage()*: returns the average calculated in the node from the values received from its neighbors.

The main method, *nextCycle()*, takes place only on the servers, which makes the simulation faster than before, since not all the nodes are working every cycle.

The actions executed in this class depend on if the *Overload Algorithm* is being executed or not. We assume that it is so we can explain the entire code.

When the load of a server is under the *availability threshold* it can help a neighbor. It takes one randomly out of its list and checks if it is busy. If not, it sends an availability message by calling *sendAvailable()*. If the answer to its message is not null, the node checks how many of the clients currently connected to the neighbor can be transferred to itself without becoming overloaded. This is notified again to the neighbor by invoking *getLoad()*. Finally, the neighbor is marked as free again by setting *busy* to false (*setBusy(false)*).

All the servers have an internal count to know how many cycles they have been in overload without receiving any availability notification. This counter is called *cycles_in_overload* and gets incremented each cycle the node is high loaded. Any time an overloaded server receives a help message from a neighbor, this counter resets.

When a node is in high load state and the value of *cycles_in_overload* is over 5 the second part of the *Overload Algorithm* starts. After some cycles without any notification, the node assumes that it does not have any neighbors available to help so it checks if any of its neighbors is sleeping. If it finds a standby neighbor, it tells the clients that make the server overloaded to reconnect to the standby node (by calling the method *getLoad()*). This way, the neighbor gets waken up and the servers stops being overloaded to be in medium load state. *cycles_in_overload* is then set to zero.

5.4.3 Reports.java

As we can tell by the name of the class, *Reports.java* is responsible for elaborating the reports containing the results of the simulations.

It has just one method, *execute()*, in which, cycle after cycle, the data is collected from the nodes and printed into the different files.

The reports we can obtain from this class are:

- *Connections_XXXXX.txt*: average number of connections per server.
- *Power_XXXXX.txt*: power consumed by all the servers.
- *Load_XXXXX.txt*: nine values per cycle showing the percentage of servers over the total size which have its number of connections in a certain interval.
- *Execution_Time_XXXXX.txt*: average execution time of the servers that are not in standby.

5.4.4 LoadPeak.java

This controller has been created as a following step to the initial situation, in which the load is constant during all the simulation. When this controller is active, a peak of load is generated.

On its main (and only) method, *execute()*, 100 new contracts are created between each client and a random server each cycle during 10 cycles (between cycles 51 and 60), having created 1,000 new contracts per client after cycle 60.

From cycles 211 to 220 the initial situation is reestablished, deleting from the servers a number of contracts equal to that created in the previous step. The same number of contracts is removed from each server. In case a server has fewer contracts than those we try to delete, the next one will delete also those contracts the former node could not.

5.4.5 Configuration file

The configurable parameters are (besides those from *PeerSim*):

- *Process capability* of the nodes.
- *Overload threshold*: number of connections from which the node considers itself overloaded.
- *Available threshold*: number of connections under which a node is considered available to help its neighbors.
- Type parameters *type_0* and *type_1* indicate the number of clients and servers respectively. The values are given as a percentage over the size of the network.
- Load parameters *low*, *medium* and *high* determine the number of connections established by every client. The values are given as a percentage over the number of clients.

The *boolean parameters* that allow us to decide if the algorithm will be executed and which reports will be generated are:

- Execute *Overload Algorithm* (*protocol.alg.overload*).
- Generate connections report (*control.rep.connections*).
- Generate power report (*control.rep.power*).
- Generate load report (*control.rep.load*).
- Generate execution time report (*control.rep.execution_time*).

5.5 Classes for load balancing and energy saving

The code remains the same as that in *5.4*, but now the class *Algorithms.java*, besides the *Overload Algorithm*, includes the behavior for the *Standby Algorithm*.

5.5.1 Algorithms.java

Just one method has been modified from the code in *5.4.2*:

- *sendAvailable(int st)*: if the receiver of an availability message is low or medium loaded and the *Standby Algorithm* is being executed, these method returns a list containing all of the clients with which it has a link. The rest of the conversation remains the same as it was before.

  The new parameter *st*, which represents the state of the sender of the message, is used to avoid the interaction between two medium load nodes.

5.5.2 Configuration file

The only difference with the configuration file in *5.4.5* is the introduction of the *boolean parameter* that tells the system if the *Standby Algorithm* has to be executed or not (*protocol.alg.standby*).

6. Simulation results

6.1 Energy saving with constant load

6.1.1 Scheduled simulations

In *Table 5* we can see the parameters of the different simulations we analyze in this section. We do not include all the configurable parameters, but those who are changed to create the different scenarios. In all the cases, the network is composed by 1,000 nodes with a *process capability* of 10 tasks/cycle. For experiments 1 to 4, they are initially assigned a random load value between 23 and 33 tasks, corresponding to the low load state. The *normal time* is always set to 100%. Overload control is local in all the simulations.

When the algorithms are not executed, with all the nodes initially in low load (average load of 28 tasks) and the said process capability, we have an execution time of 2.8 cycles.

In the table, a cell with multiple values represents simulations where we only change the parameter corresponding to that cell, while the rest of them remain the same.

The number in the first column assigns a number to each experiment so we can identify them.

All the simulations have been repeated 10 times. After that, the average of the values obtained is calculated by using the class *Average,* which has been defined outside the *PeerSim* environment. The charts show the results corresponding to that average.

Experiment number	Initial load	Number of neighbors	Matchmaker threshold	Matchmaker limit	Slow time
1	All the nodes in low load	20	5%	10	*300%*
					200%
					175%
					150%
					125%
					100%
2	All the nodes in low load	20	*1%*	10	150%
			2%		
			3%		
			5%		
			10%		
			20%		
			50%		
3	All the nodes in low load	20	5%	*0*	150%
				5	
				10	
				20	
				30	
				50	
				100	

Experiment number	Initial load	Number of neighbors	Matchmaker threshold	Matchmaker limit	Slow time
4	Varies from all the nodes in low load to all of them in high load	20	5%	10	150%
5	All the nodes in low load	*2*	5%	10	150%
		3			
		4			
		5			
		10			
		15			
		20			

Table 5: Scheduled simulations

Finally, we have *Experiment* 6, in which we compare the results obtained in *6.1.2* with those obtained with the same parameters when we remove the matchmaker.

6.1.2 Experiment 1: Different *slow time*

In *Figure 18* and *Figure 19* we can see the results of the first simulations defined in *Table 5*. In them, we fix all the parameters but the *slow time*, which vary from 300% to 100%. As we explained before, this parameter determines at which point the node decides it is overloaded.

A value of 300% means that the node will consider itself overloaded when its *execution time* is over 13.2 cycles. With the actual implementation a node will never reach that value, so we could say no overload control is being executed in this simulation. Therefore, we can see the maximum energy saving value we could achieve if we did not care about execution time.

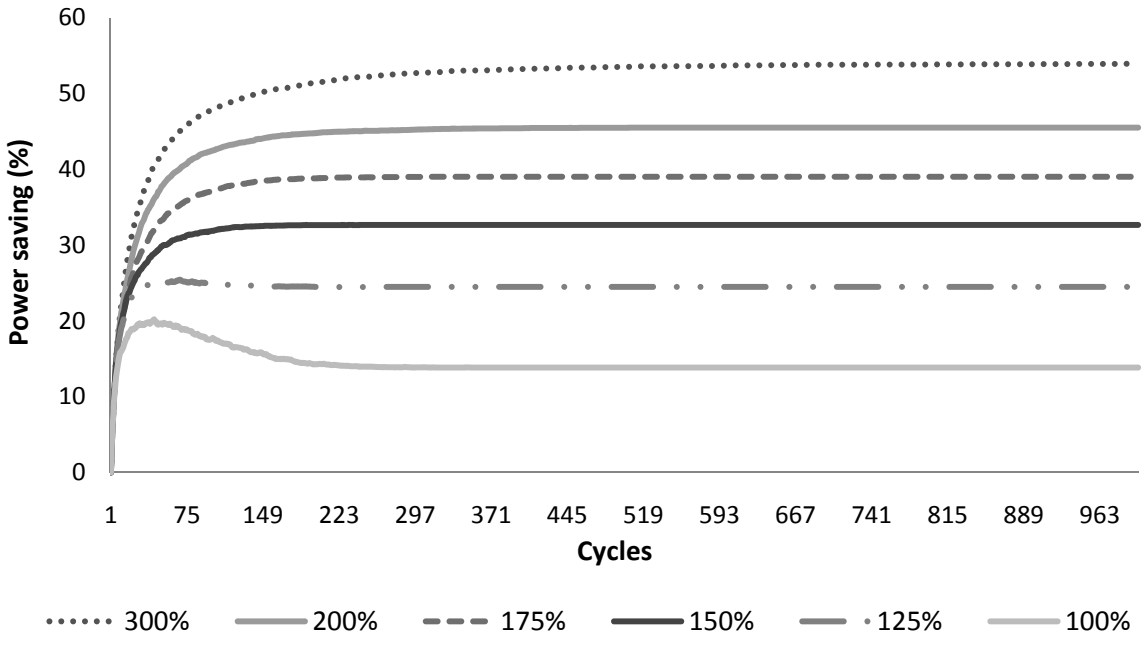


Figure 18: Power saving when the slow time varies from 300% to 100%

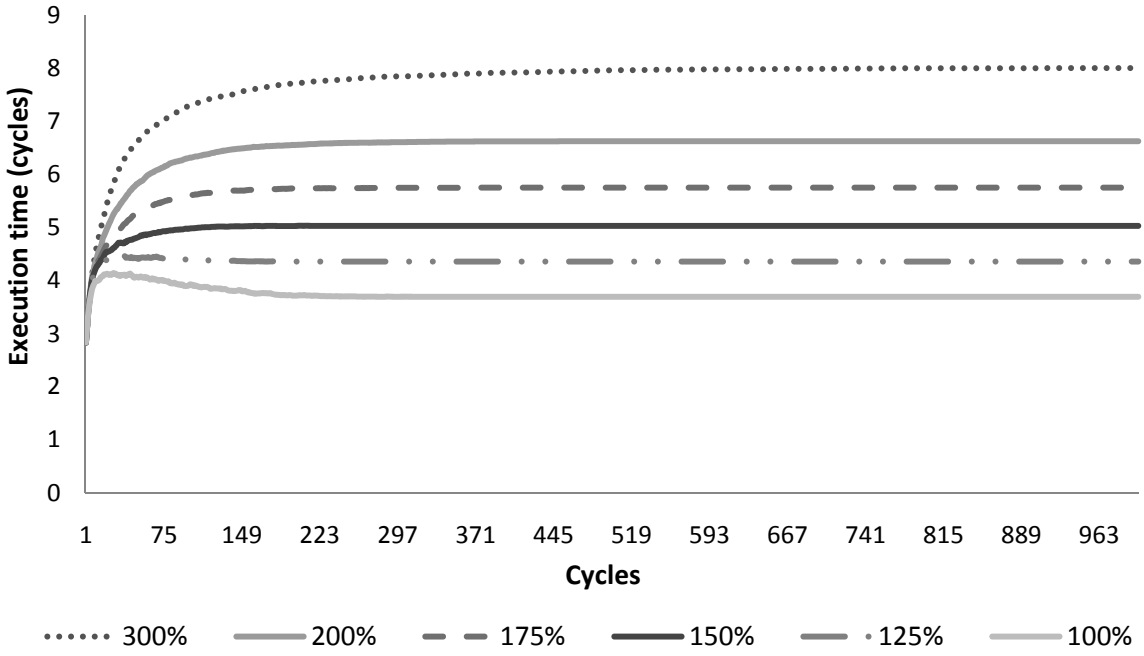
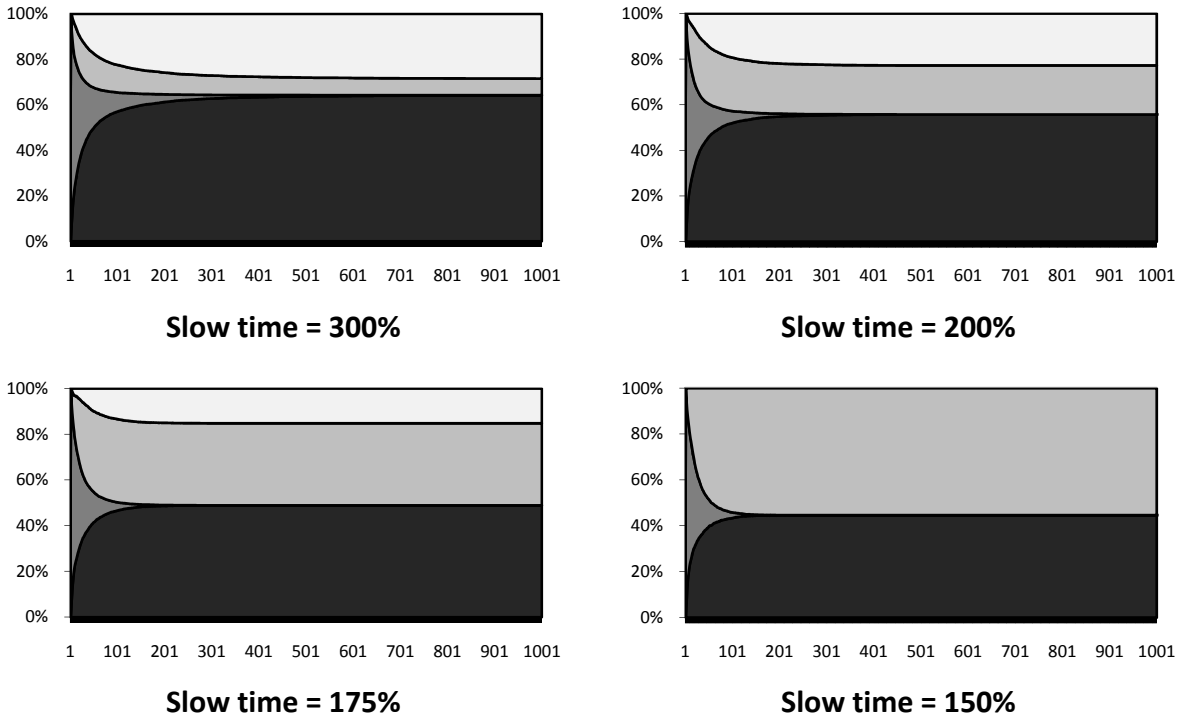


Figure 19: Execution time when the slow time varies from 300% to 100%

In all the cases we observe a very fast rising of the energy saving. As the number of low load nodes decrease (*Figure 20*), the saving slows down and converges to the final power saving value that can be achieved according to the parameters.

In *Figure 19* we can see the execution time has the same behavior described for the power saving.

When overload control is not executed (*slow time* of 300%) the final saving value is 53.93%, rising over 50% around cycle 150. Unfortunately, the average execution time in the system is also very high, close to 8 cycles.



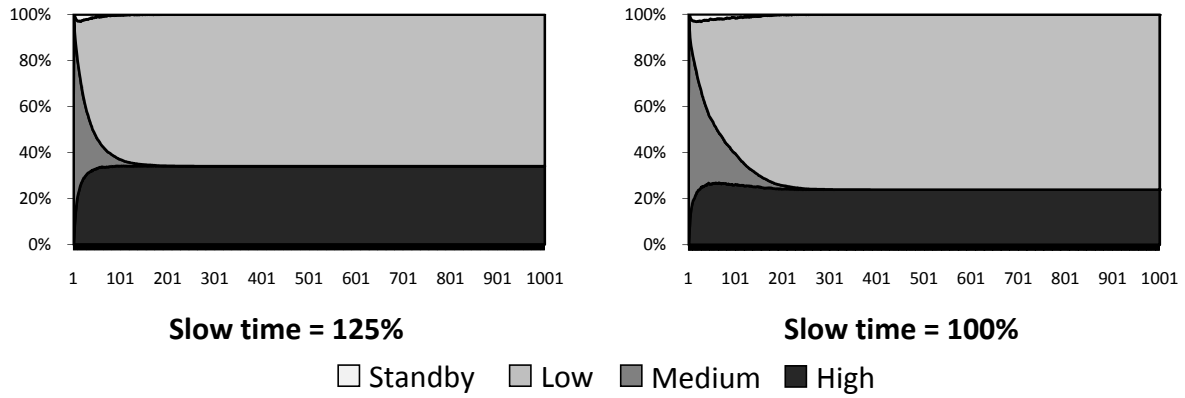


Figure 20: Percentage of nodes in every state when the slow time varies from 300% to 100% [vertical axis = nodes in the system, horizontal axis = cycles]

In *Figure 20* we see the evolution of the nodes in every state. When we have a high number of standby nodes we have high savings, but also high execution times, since the rest of the nodes have to execute the tasks that were initially assigned to the standby nodes.

With the slow time set to 300% (no overload control) we achieve the maximum number of standby nodes (64.20%), which makes us get the maximum saving. In this case we get also the maximum number of high loaded nodes (28.47%), so we have the highest execution time.

By reducing the slow time, overload control starts being applied. This results in the reduction of the high loaded nodes by waking up some standby ones. With this, the execution time decreases, but so does the power saving.

Finally, we analyze the compromise between power saving and execution time we mentioned at the beginning of this document. We will take the decision of the slow time value according to our needs and to the performance results of the different values.

In *Figure 21* we can see the trade-off corresponding to the relationship power saving – execution time. We see that for a slow time of 125% or 100% this value is considerably smaller than for the other cases, so we will not use these values for experiments 2, 3 and 4.

For the rest of values given to the slow time, we see that when the trade-off stabilizes the values achieved are very close, so any of those values would fit well for the next simulations. Anyway, since we want to keep a low execution time, for the next experiments we will use a slow time of 150%, with which we achieve a power saving of 32.66% with an execution time of around 5 cycles (from *Figure 18*).

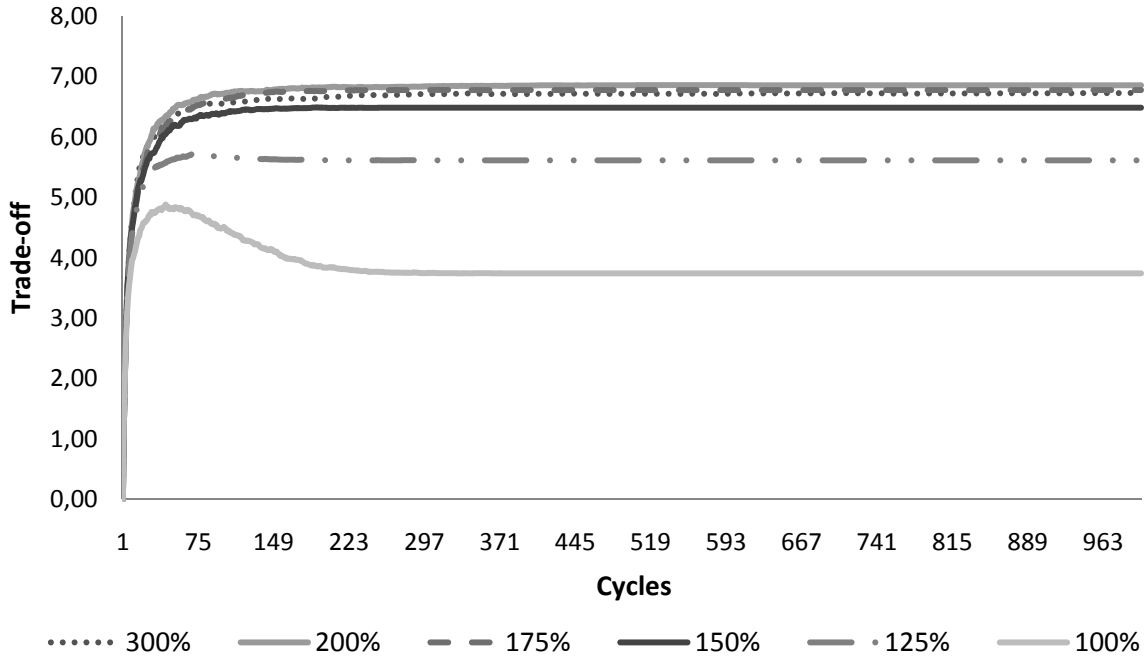


Figure 21: Trade-off power saving / execution time

6.1.3 Experiment 2: Different *matchmaker threshold*

In this second simulation we observe how results change with different values of the *matchmaker threshold*, which represents the probability of a node to act as a matchmaker. We tried with a range of values that goes from 1% to 50%.

Since this parameter represents the probability of a node to act as a matchmaker, the final results (once they stabilize) should not be different from the ones obtained in *6.1.2*, where the slow time was 150%.

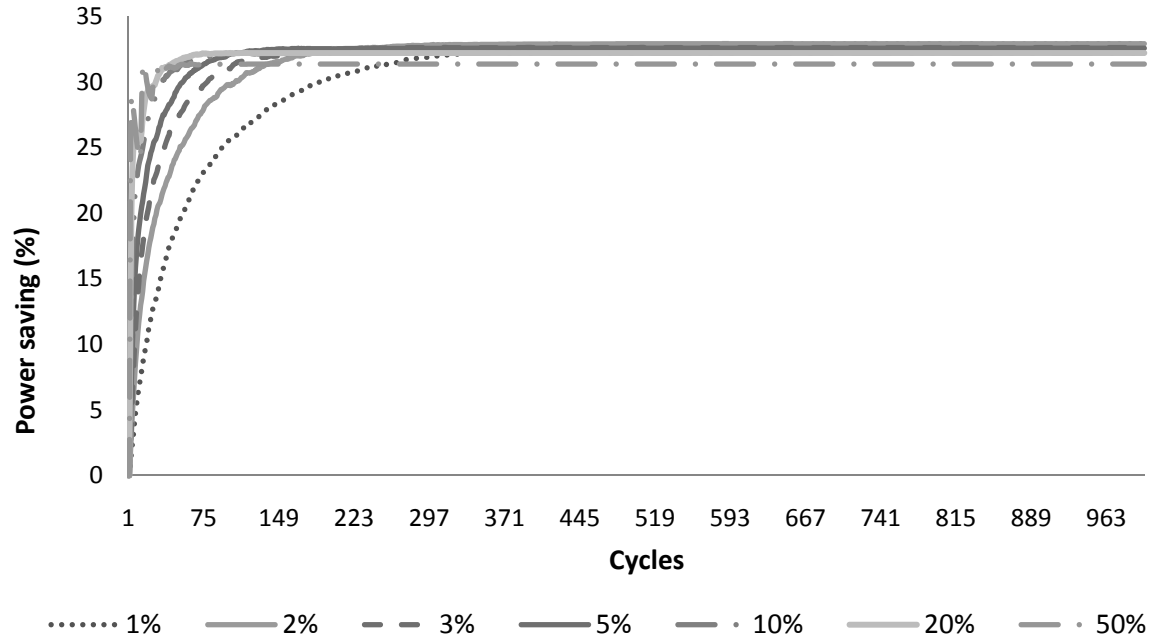


Figure 22: Power saving when the matchmaker threshold varies from 1 to 50

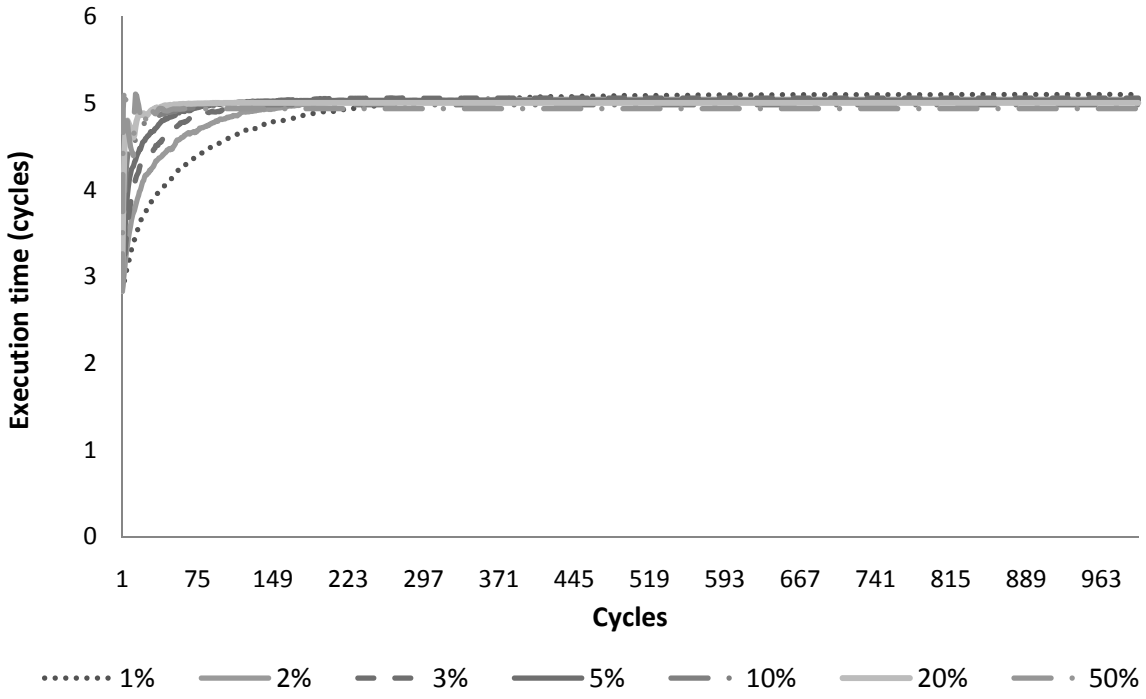
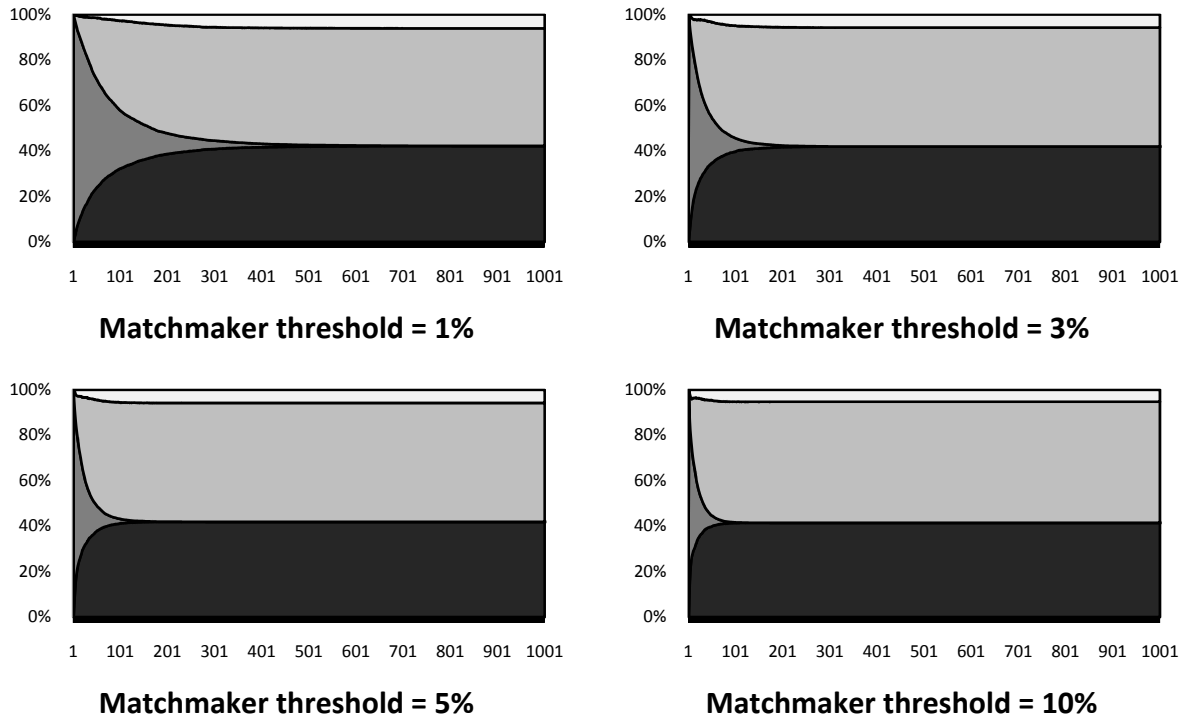


Figure 23: Execution time when the matchmaker threshold varies from 1 to 50

Analyzing *Figure 22* and *Figure 23* we confirm what we anticipated before: power saving and execution time have approximately the same values we obtained in the previous experiment with a slow time of 150% (saving of 32.66% and execution time of 5 cycles). The only important difference between the different simulations is the time the system needs to converge to the final values. This fact can be seen also in *Figure 24*, where the main difference between the different simulations is also the time it takes to reach stability.



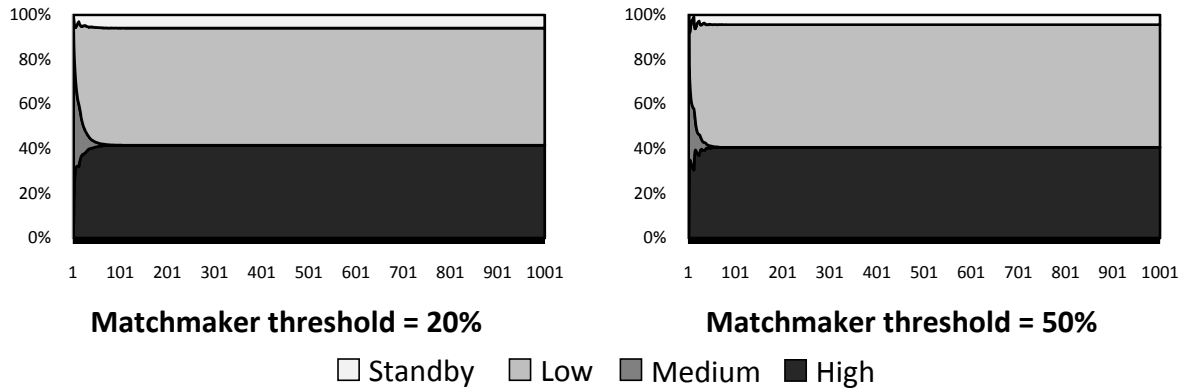


Figure 24: Percentage of nodes in every state when the matchmaker threshold varies from 1% to 50% [vertical axis = nodes in the system, horizontal axis = cycles]

Watching carefully *Figure 22*, *Figure 23* and *Figure 24* we see a slightly different behavior for a threshold of 50%. In *Figure 25* we have the first 100 cycles of its power saving line, so we can see this behavior more clearly.

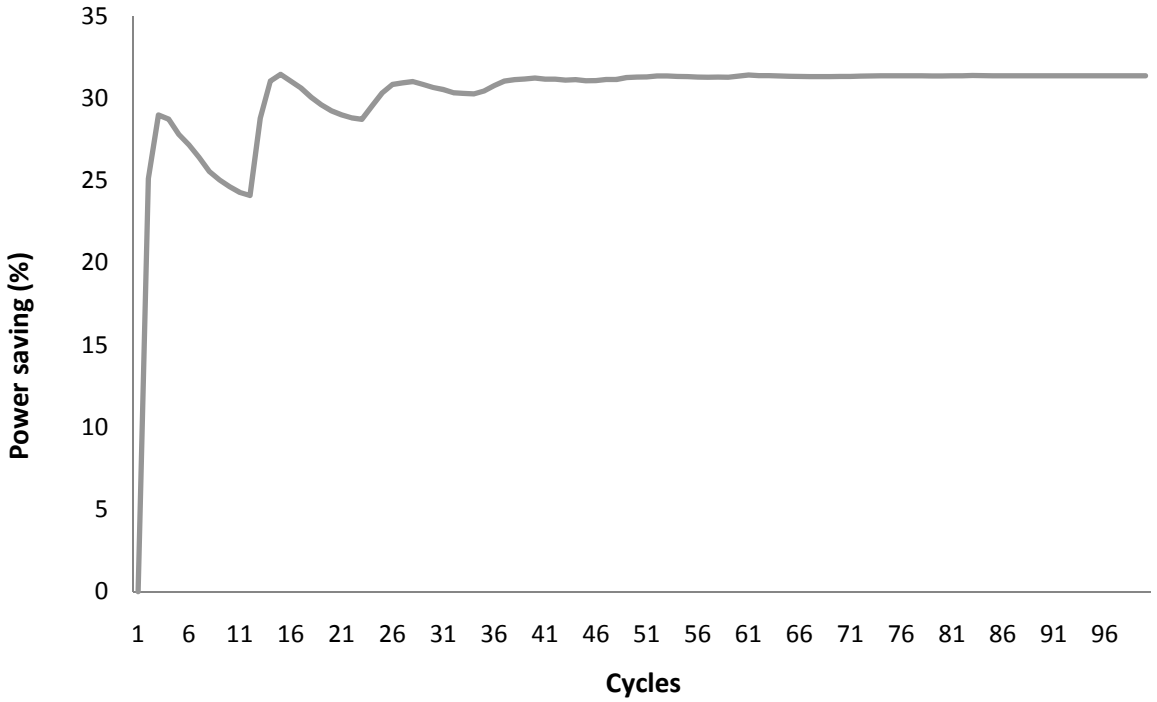


Figure 25: Detail of the power saving line corresponding to a matchmaker threshold of 50%

As we see, the power saving shows some fluctuations in the beginning. This is because the probability of acting as a matchmaker is so high that we get huge saving values in the very beginning, but the execution time also rises very quickly. Then, the *Overload Algorithm* wakes up the standby nodes necessary to avoid the overload situation, reducing the saving (and the execution time). The number of cycles between each of the peaks shown in the chart depends on the *matchmaker limit* parameter (in this case 10), as we will see in the next simulation.

Since the values are very similar for every value of the matchmaker threshold it is useless to include the chart showing the relationship between power saving and execution time, because it is

clear that the only difference between the lines will be the speed of convergence to the final value (which will be more or less the same in all the cases).

We could conclude the analysis of this experiment by saying that the results will be the same independently of the matchmaker threshold, so we could choose this value freely. However, a higher matchmaker threshold would mean that the nodes act more often as a matchmaker which, even though is not considered in this model, we can think that would entail a higher power consumption. For this reason, for the next experiments we will choose a value that is not too high, like 5%. Anyway, the decision relies again on our needs.

6.1.4 Experiment 3: Different *matchmaker limit*

The next parameter we analyze is the *matchmaker limit*: the number of cycles a node has to wait after acting as a matchmaker before it can do it again. The results, shown in *Figure 26* and *Figure 27*, are very similar to those in *6.1.3*, with the same final values but different convergence times.

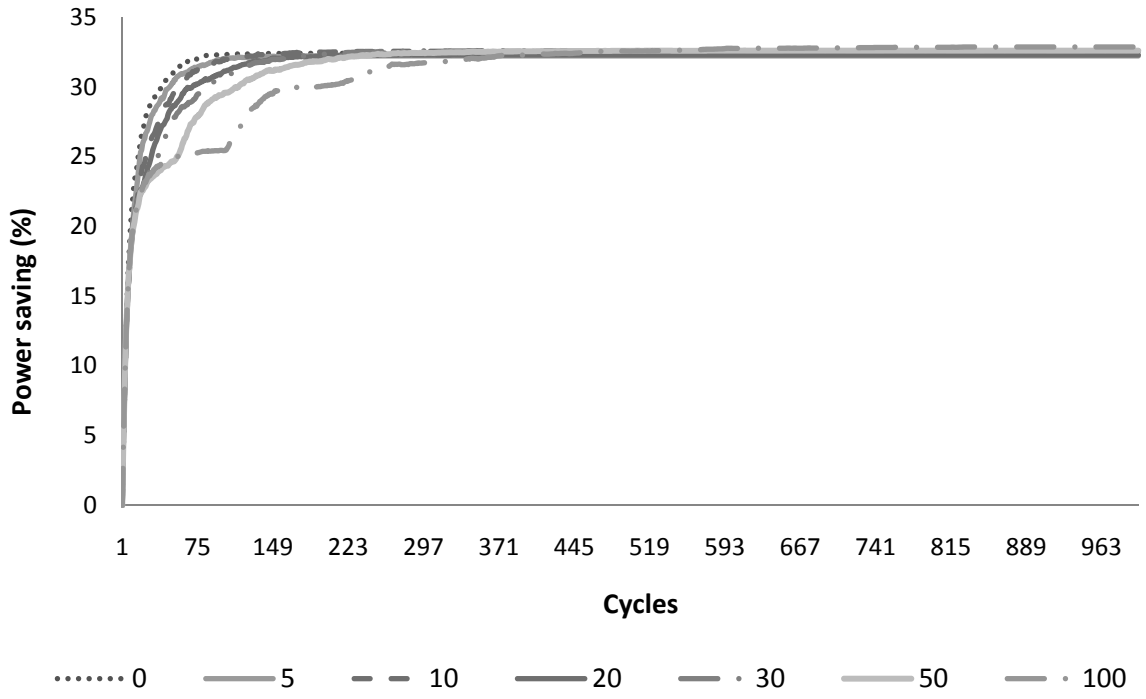


Figure 26: Power saving when the matchmaker limit varies from 0 to 100

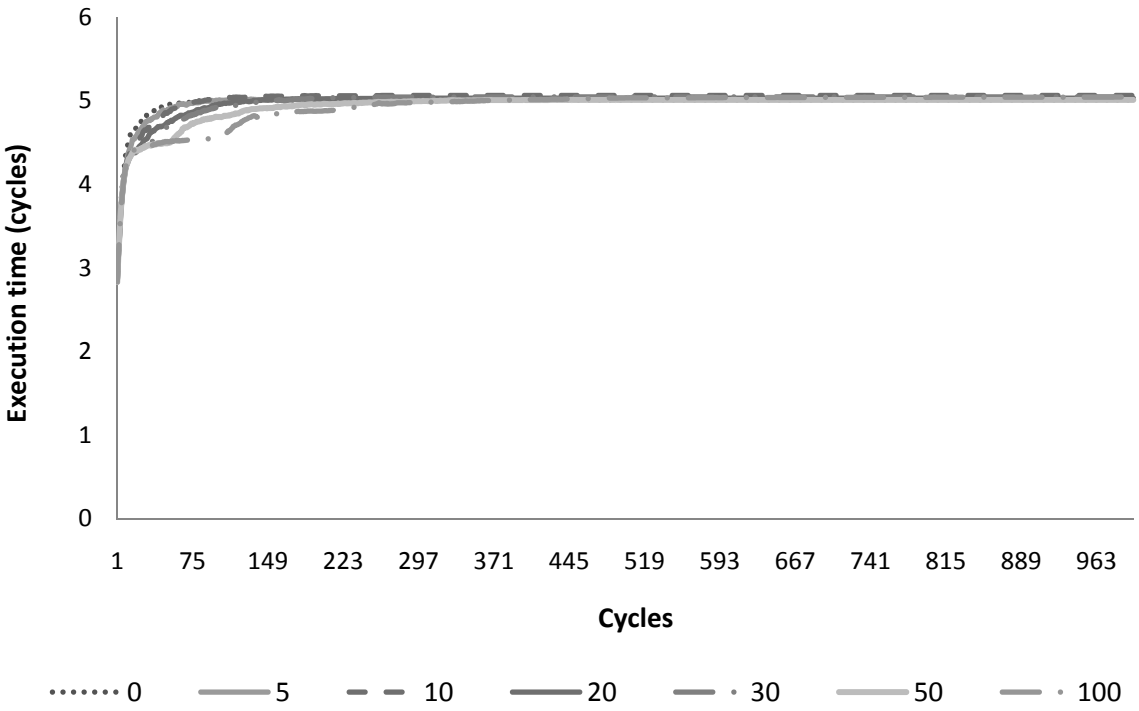
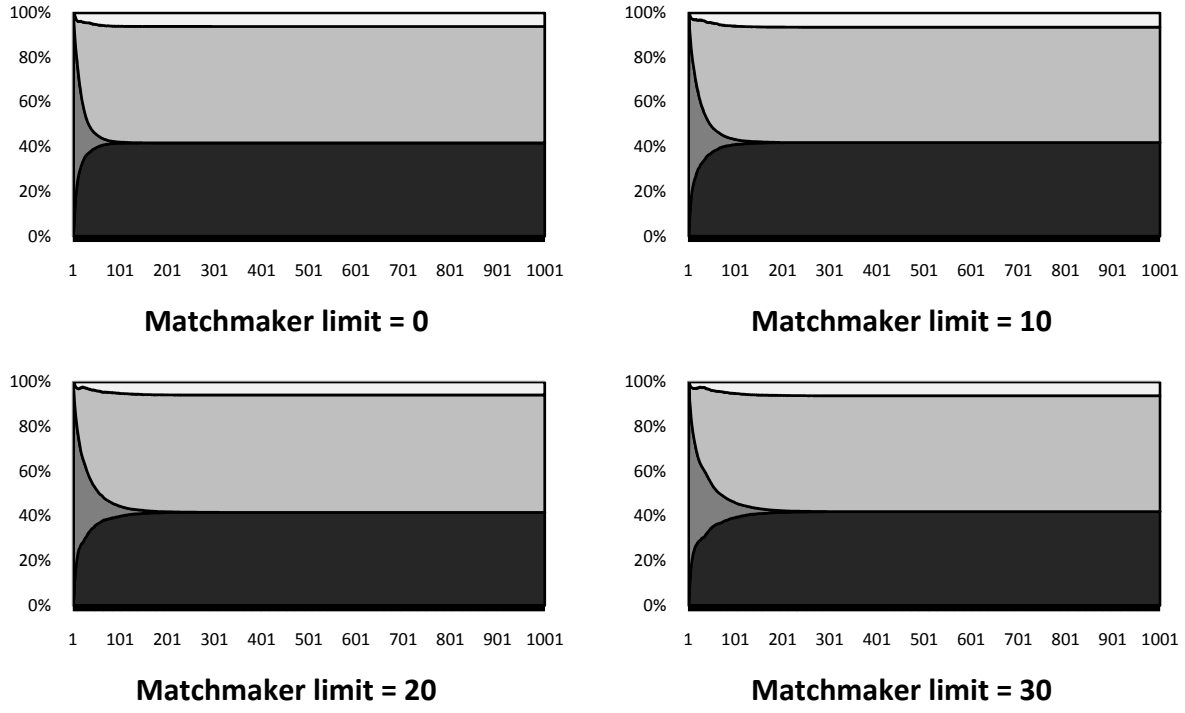


Figure 27: Execution time when the matchmaker limit varies from 0 to 100

As we wrote in the analysis of the previous experiment, the length of those fluctuations we could see when the *threshold* was 50% is determined by the value assigned to the *limit*. In this case, with a *threshold* of 5%, we can clearly see this when the *limit* is 100 (green line).

At first all the nodes can act as a matchmaker, so there is a fast rising. After this, the nodes have to wait 100 cycles until they can be the matchmaker again. That is why the value stops rising until the *limit* is reached, when the *Standby Algorithm* can be executed again.

This effect can also be seen in the last chart included in *Figure 28*, corresponding to the matchmaker limit of 100.



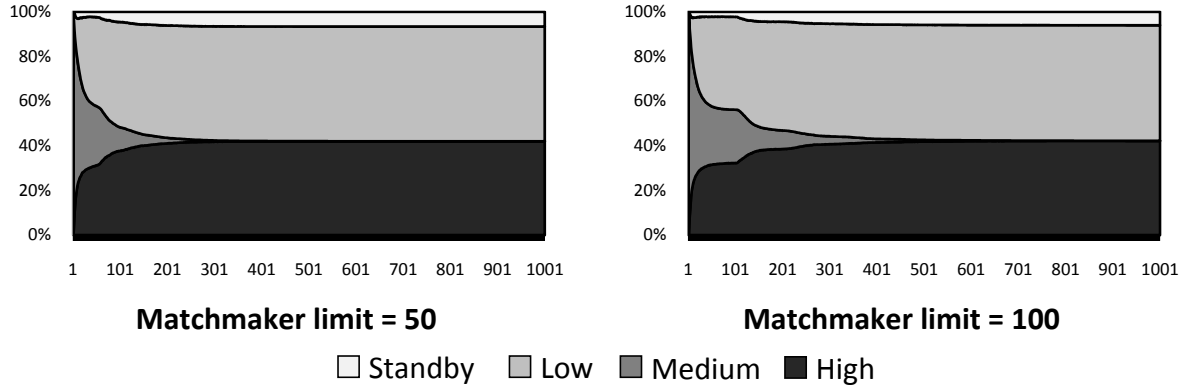


Figure 28: Percentage of nodes in every state when the matchmaker limit varies from 0 to 100 [vertical axis = nodes in the system, horizontal axis = cycles]

6.1.5 Experiment 4: Different initial situations

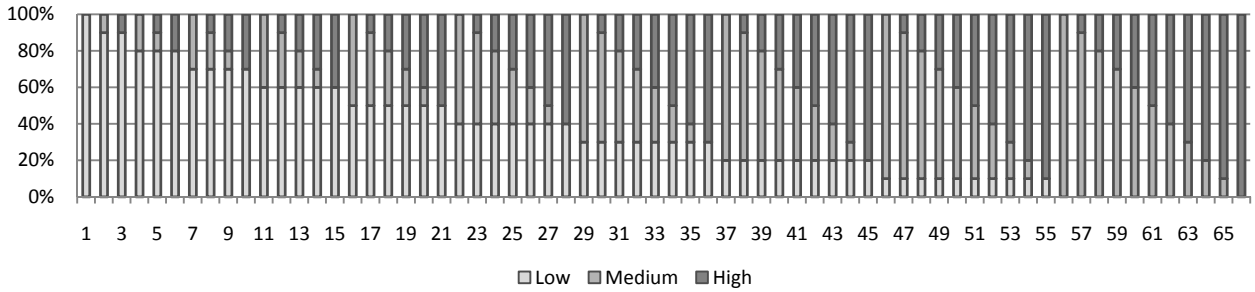


Figure 29a: Different initial situations [vertical axis = nodes in the system, horizontal axis = number of sub-experiment]

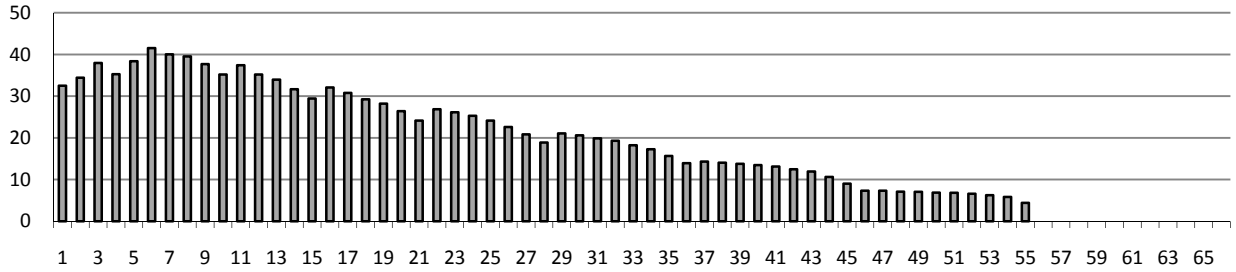


Figure 30b: Power saving with different initial situations [vertical axis = power saving (%), horizontal axis = number of sub-experiment]

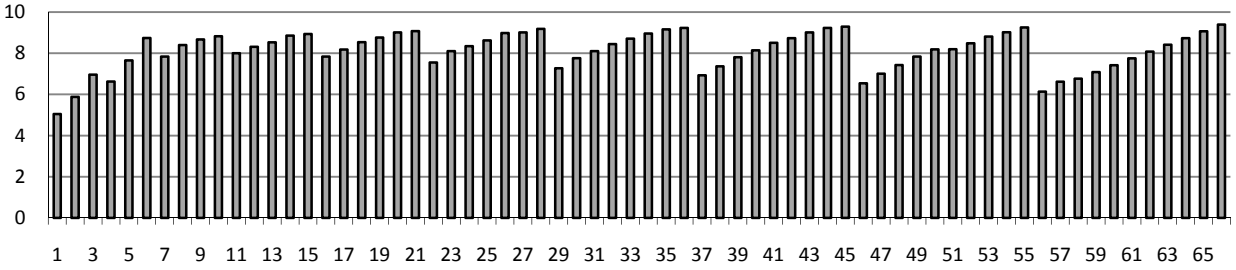


Figure 31c: Execution time with different initial situations [vertical axis = execution time (cycles), horizontal axis = number of sub-experiment]

Figure 29 shows the power saving achieved with different initial situations and a slow time set to 150%. The x-axis represents 66 different simulations, each one of them with the initial conditions shown in *Figure 29a*. *Figure 29b* shows the maximum power saving achieved in every case and *Figure 29c*, the maximum execution time.

We observe that the maximum saving values are reached in the first cases, when most of the nodes are initially low loaded. This is exactly what we expected, since our algorithm works with the nodes in that state. When there are initially a few or no nodes with low load the energy saving tends to zero.

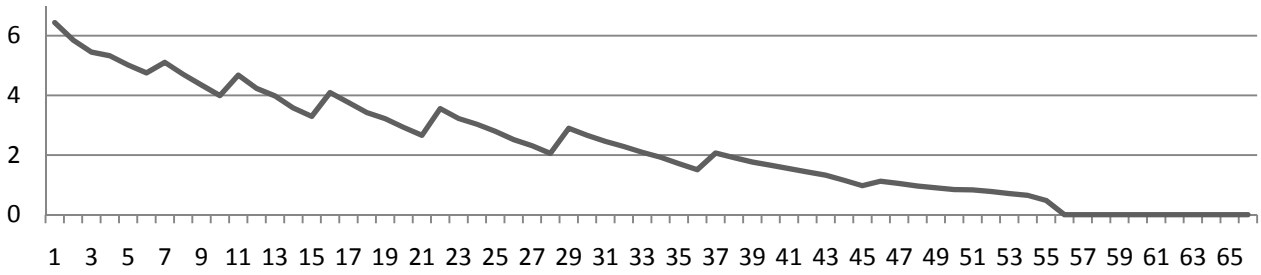


Figure 32: Trade off power saving / execution time
[vertical axis = trade off, horizontal axis = number of sub-experiment]

It is also important to consider the execution time, since the efficiency of the system consists, as we have said before, on a compromise between the execution time and the power saving. This relationship can be seen in *Figure 30*, where we observe that the best ratio saving/execution time is achieved when all the nodes are initially low loaded. We can see also that the trade-off rises a little every time that we remove the high loaded nodes from the initial conditions (simulations 7, 11, 23, 29, 37 and 46).

6.1.6 Experiment 5: Different number of neighbors

In this experiment we compare the power saving (*Figure 31*) and execution time (*Figure 32*) values obtained when we change the number of links established by each node. The only line separated from the rest is that corresponding to N=2. For the other values of N the results are more or less the same.

This means that there is almost no difference between having 6 or 40 neighbors (remember *4.1.1*, where we explained that the number of neighbors is usually 2·N). Knowing that, in a real implementation we should reduce the number of neighbors in order to save some resources. In these simulations we did not consider the cost of the creation and maintenance of the overlay, but it is obviously not the same having 40 neighbors per node than having 6.

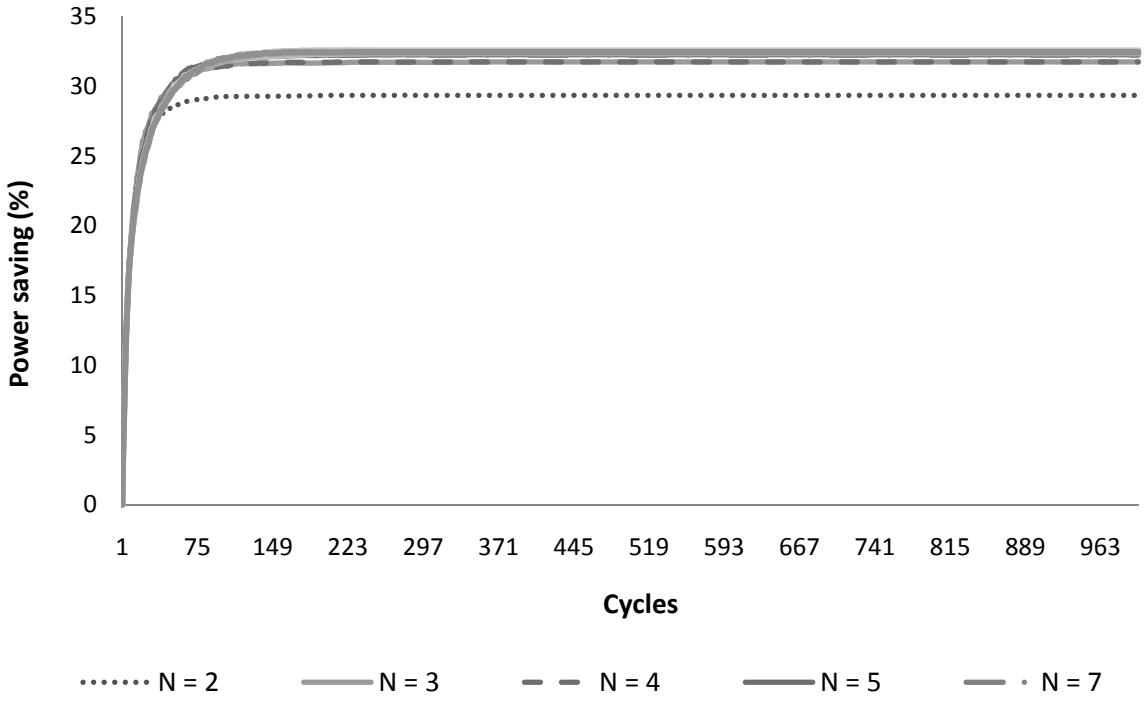


Figure 33: Power saving with different number of links

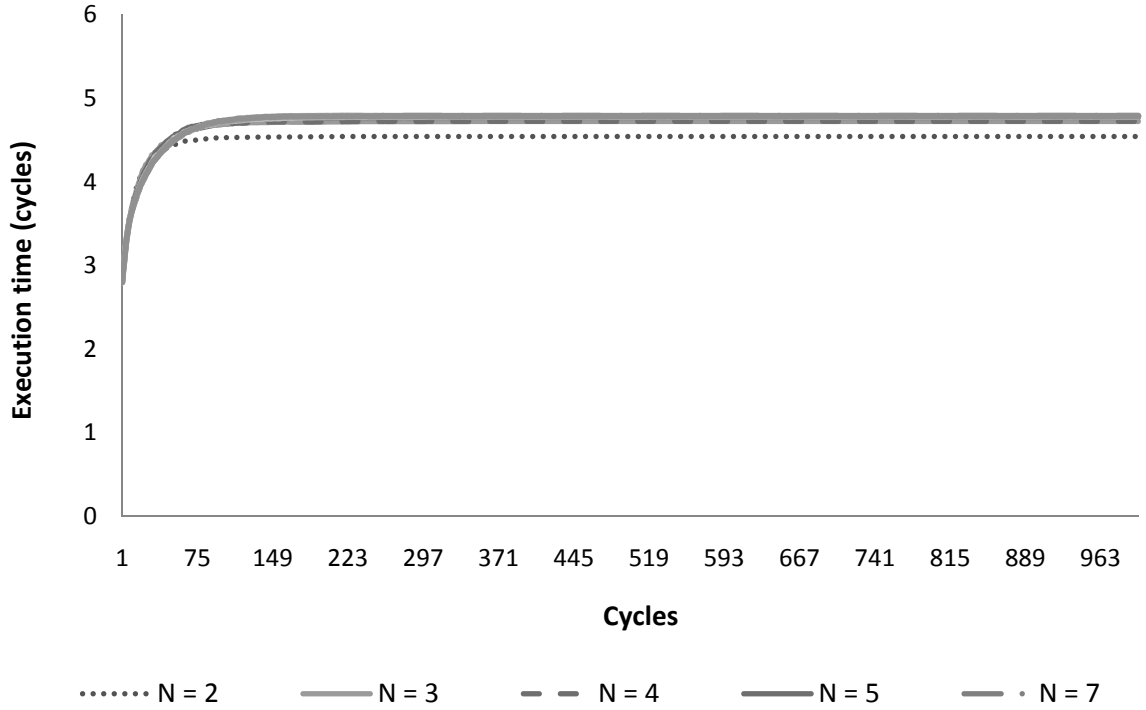


Figure 34: Execution time with different number of links

6.1.7 Experiment 6: Eliminating the matchmaker

Finally, we have made some simulations comparing the energy saving and execution time obtained with the figure of the matchmaker and without it (the nodes directly contact their neighbors and exchange their load).

While *Figure 33* and *Figure 34* show the charts we already saw in *6.1.2* (where we had the matchmaker), in *Figure 35* and *Figure 36* we find the results we obtain when we remove the matchmaker (with the same parameters).

We see that the charts in the two cases are practically equal. Seeing that, it seems logic to choose the second option, since it is faster and consumes fewer resources.

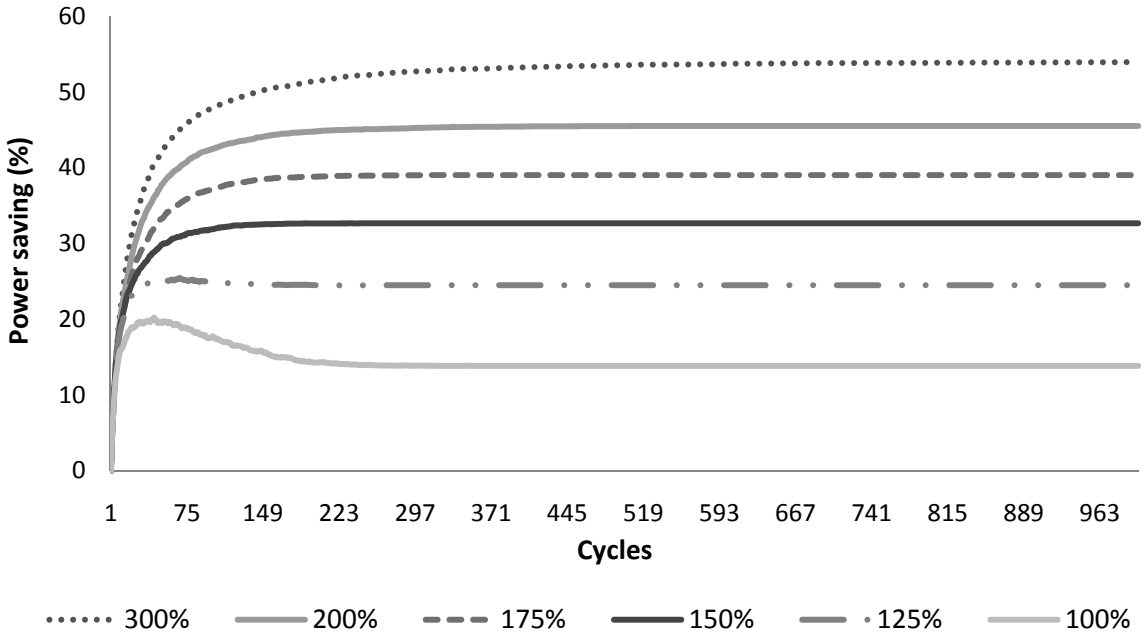


Figure 35: Power saving with different slow time and the figure of the matchmaker

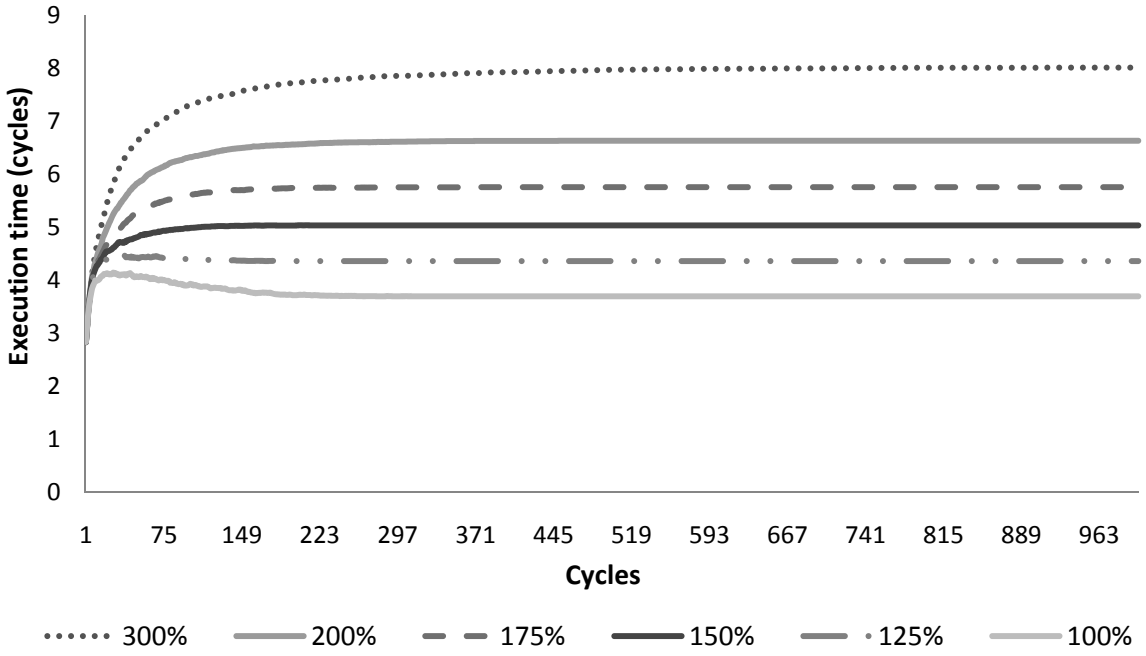


Figure 36: Execution time with different slow time and the figure of the matchmaker

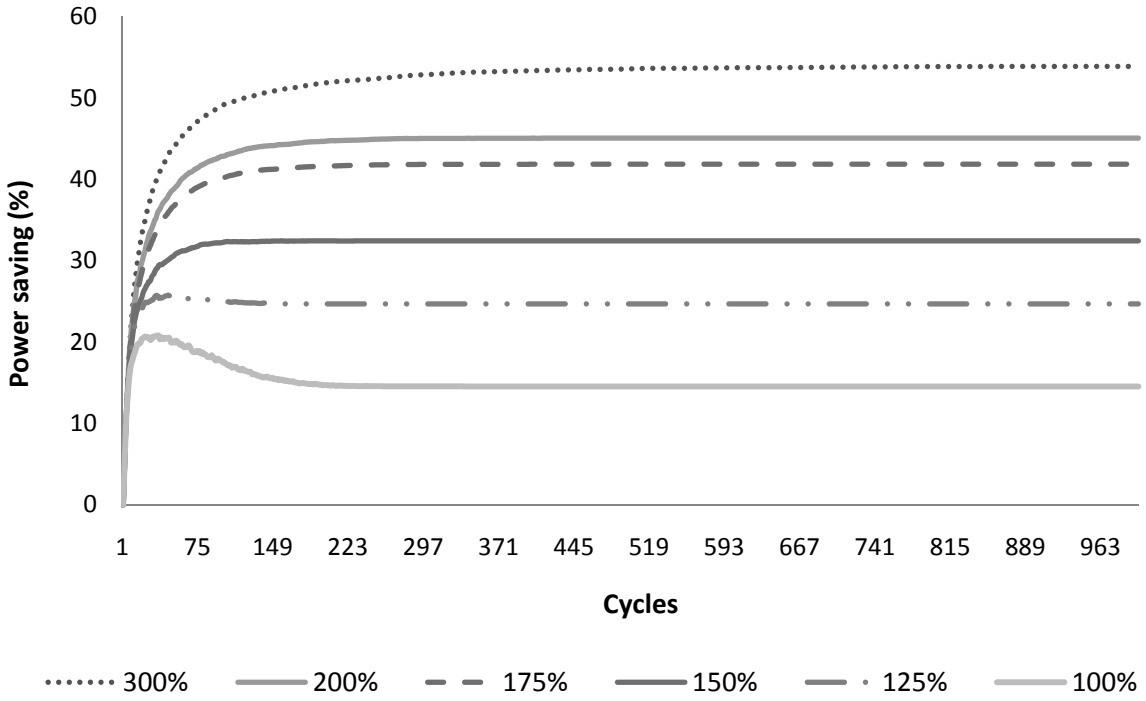


Figure 37: Power saving with different slow time and without the figure of the matchmaker

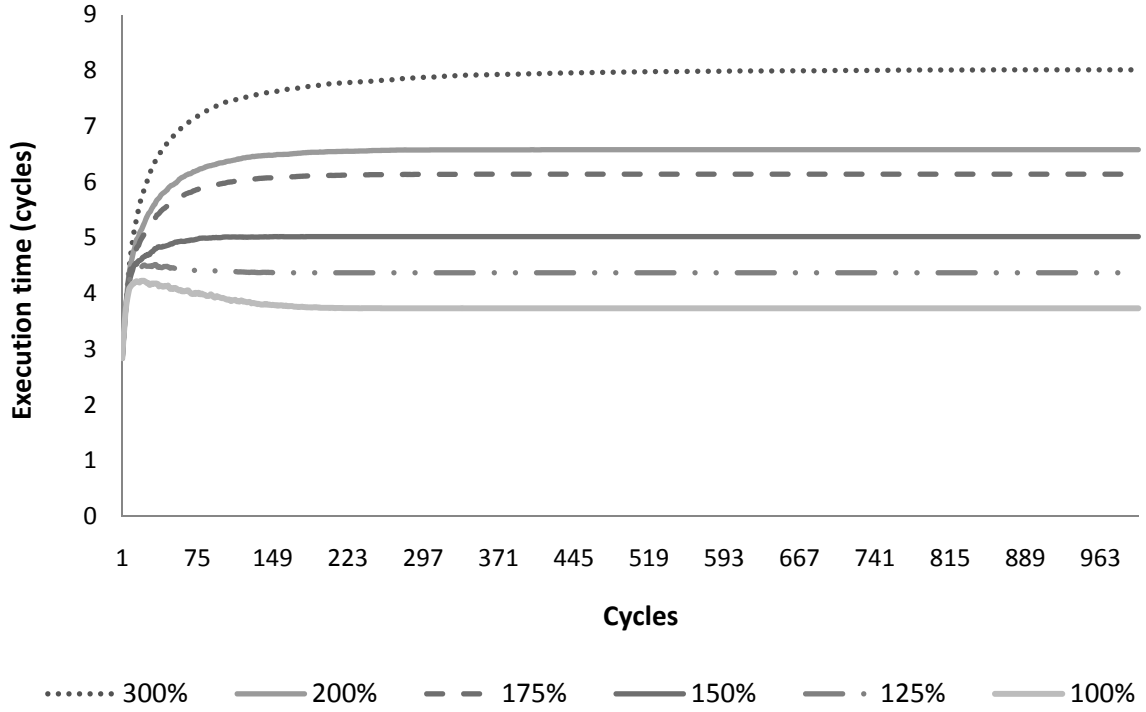


Figure 38: Execution time with different slow time and without the figure of the matchmaker

6.2 Energy saving with variable load

6.2.1 Scheduled simulations

Since we want to simulate the time of a day, the number of *cycles* is set to 1,500 (as we said before, one day has 1,440 minutes, so 1,500 is a good value for this parameter).

As we decided after analyzing the simulations of the first model, we set the matchmaker *threshold* to 5 and the *limit* to 10. We run all the simulations with the whole system initially in low load, so we can see the best performance our model can offer.

In this section we have two experiments:

- *Experiment* 1: Fixed number of neighbors (20): analysis of the performance of the model when the number of neighbors is set to 20.
- *Experiment* 2: Different number of neighbors: comparison of the results obtained with different number of neighbors (2, 3, 4, 5, 10, 15 and 20).

All the simulations have been repeated 20 times, the charts showing the average values of those repetitions.

6.2.2 Experiment 1: Fixed number of neighbors (20)

First of all, we need to run a simulation where none of the algorithms is executed. This way we will get the reference values, corresponding to the state of the nodes and the power consumed by them during a day. When we run later simulations we will compare them to this one to calculate the power saving.

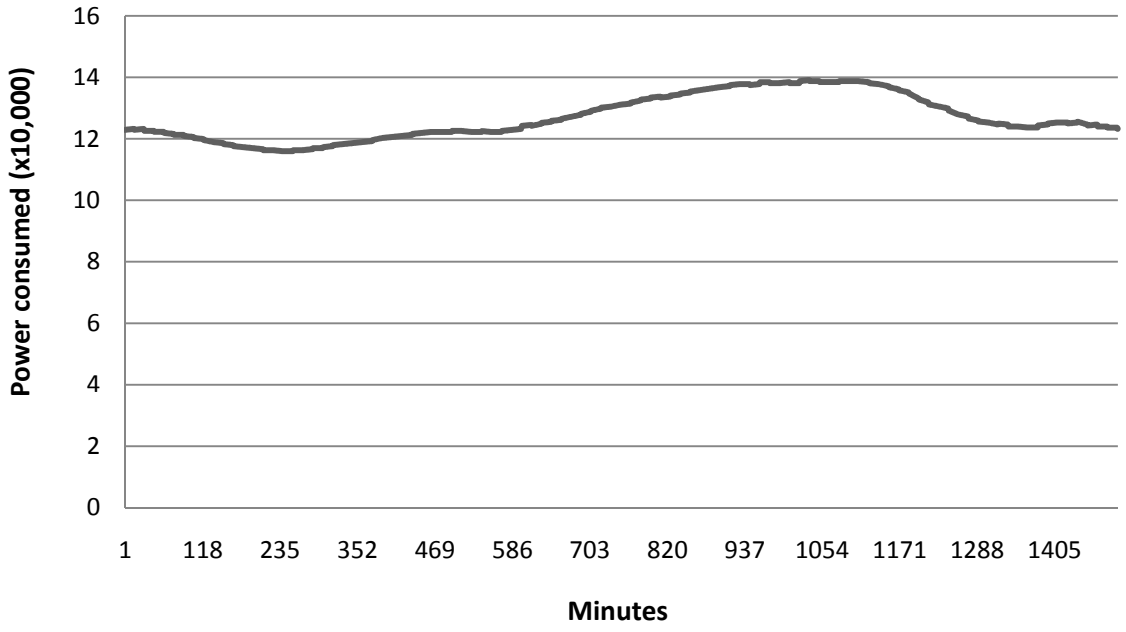


Figure 39: Power consumed by the system throughout a day

Figure 37 shows the power consumed by the system during the different periods of the day. We notice that the shape is obviously the same as that in *Figure 17*, since the algorithms are not being executed in this simulation.

With this as the starting point, we can now see if we can improve the results by introducing the algorithms we described before.

To apply these algorithms it is necessary to set the thresholds *low_limit* and *high_limit*. Since the former does not change much the results, we fixed it to 10,000 connections, while we try four different values for the latter: 150,000, 125,000, 100,000 and 80,000 connections.

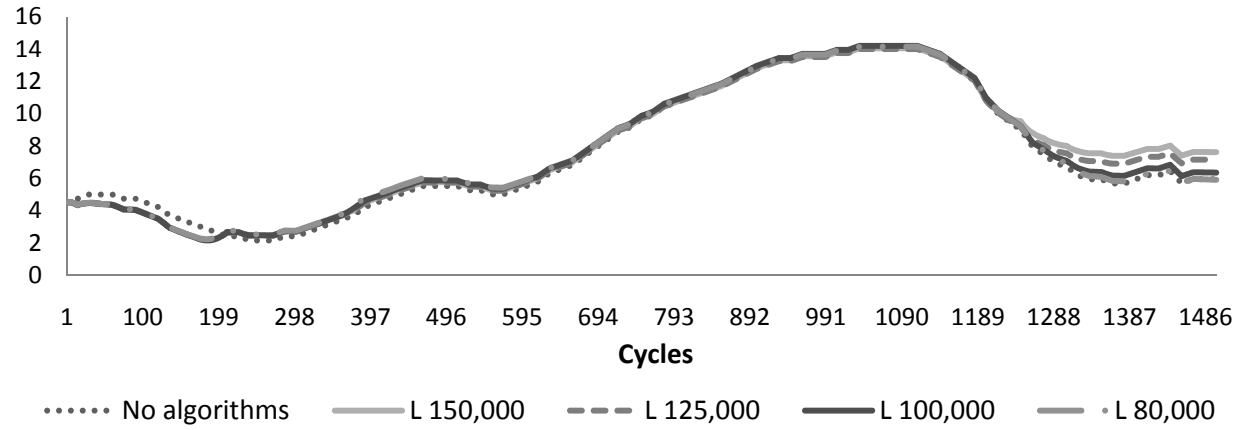


Figure 40: Number of connections in the different simulations
[vertical axis = number of connections x10,000]

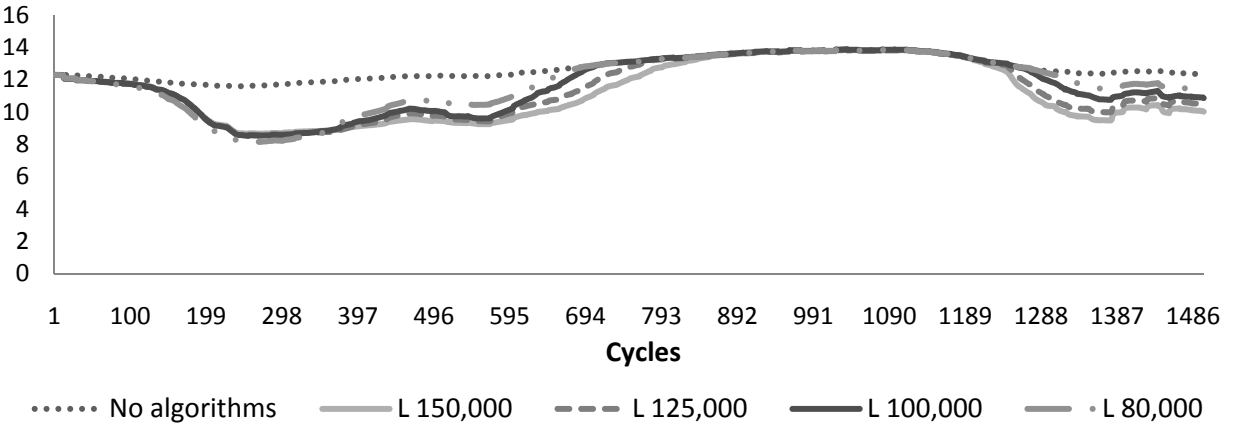


Figure 41: Power consumed in the different simulations
[vertical axis = power consumed x10,000]

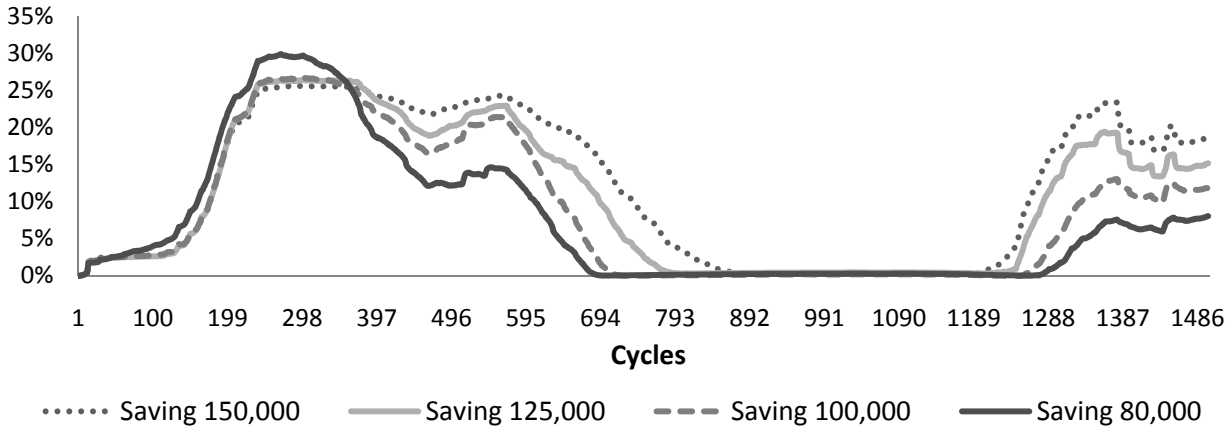


Figure 42: Energy saved in the different simulations [vertical axis = % of power saving]

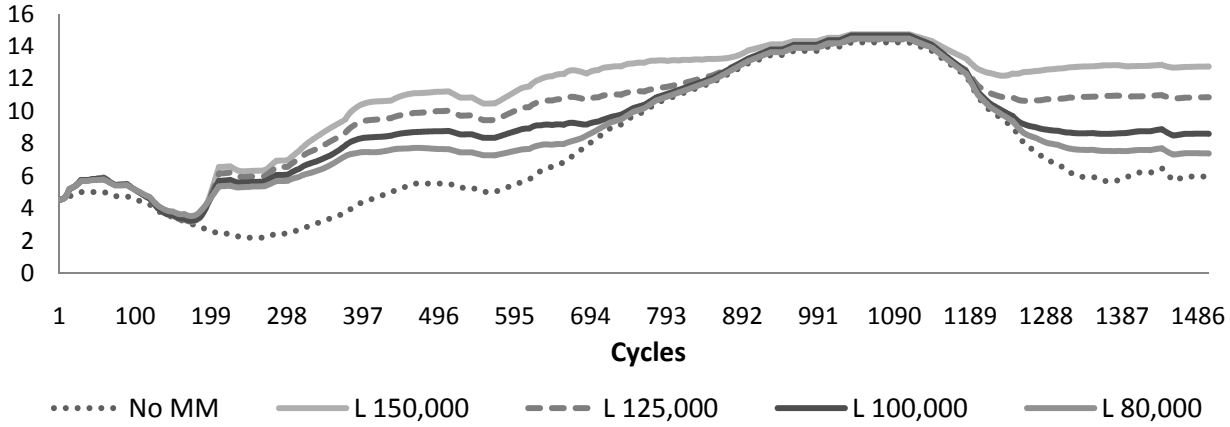


Figure 43: Execution time in the different simulations [vertical axis = execution time in cycles]

We can see from *Figure 38* that the number of connections is practically the same through the different simulations, as we expected, since the service offered by the system should be the same as in the first situation. Therefore, this chart is not of special interest, but we included it because it is helpful to analyze *Figure 39*, *Figure 40* and *Figure 41*.

Figure 39 shows the power consumption by the whole system in the different simulations, depending on the value given to *high_limit*. In *Figure 40* the power saving achieved in the different simulations is presented. We see that with a bigger limit we get better saving values, but this means there are more high loaded nodes, since they do not considered themselves overloaded until the limit is reached. This fact can be seen on *Figure 42*, where the percentage of nodes in each of the states throughout the day is shown. We notice that as *high_limit* decreases we have a better distribution of the load, so the execution time also decreases (shown in *Figure 41*).

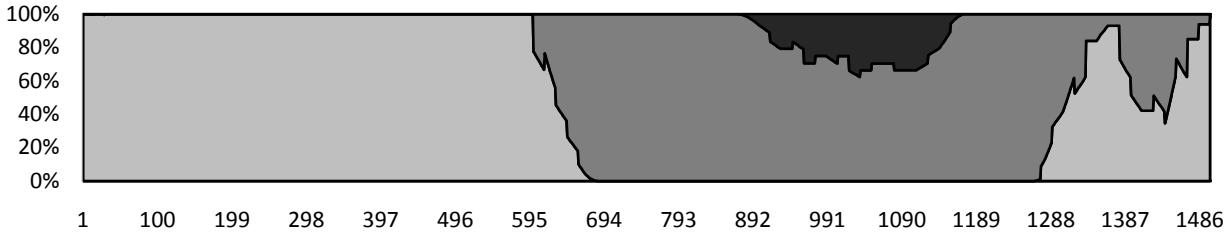


Figure 44a: Percentage of nodes in every state with no algorithms
[vertical axis = nodes in the system, horizontal axis = cycles]

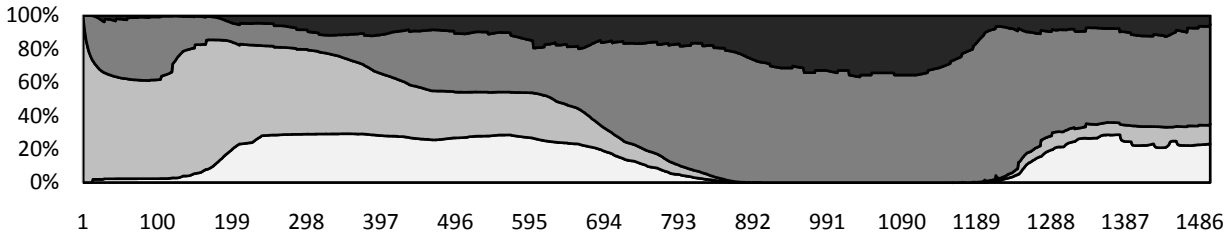


Figure 45b: Percentage of nodes in every state with *high_limit* = 150,000
[vertical axis = nodes in the system, horizontal axis = cycles]

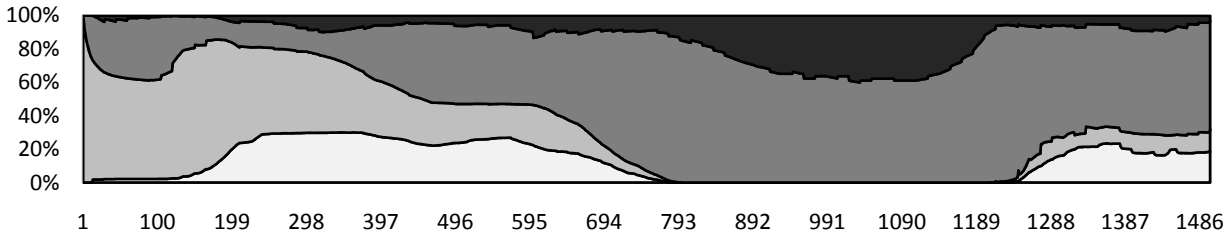


Figure 46c: Percentage of nodes in every state with *high_limit* = 125,000
[vertical axis = nodes in the system, horizontal axis = cycles]

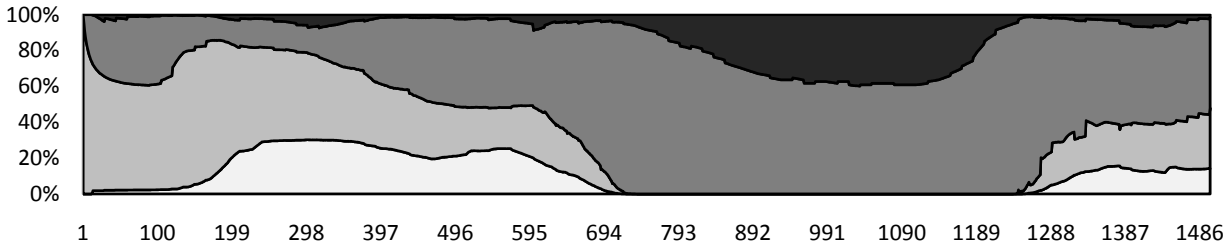


Figure 47d: Percentage of nodes in every state with *high_limit* = 100,000
[vertical axis = nodes in the system, horizontal axis = cycles]

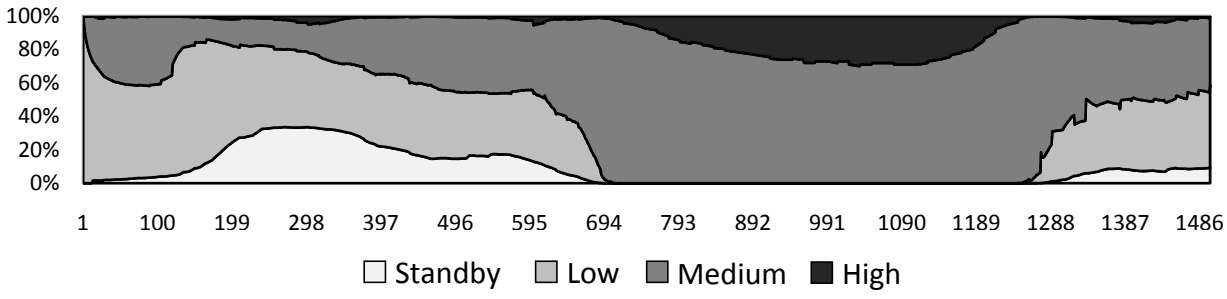


Figure 48e: Percentage of nodes in every state with *high_limit* = 80,000
[vertical axis = nodes in the system, horizontal axis = cycles]

If we calculate the total power consumed by the system in the different scenarios and we compare it with the consumption when the system is running without any additional algorithms, we obtain the total saving values. *Table 6* shows this percentage of energy that can be saved in a day, along with the average execution time [36].

high_limit	Total saving	Average execution time
150,000	11.66%	11.04 cycles
125,000	10.07%	10.09 cycles
100,000	8.23%	9.33 cycles
80,000	7.03%	8.83 cycles

Table 6: Percentage of energy saved depending on the value of *high_limit*

Finally *Figure 43* shows the relationship between power saving and execution time. We see that, for the different values assigned to the *high_limit*, the results obtained are very close to each other. This means that this parameter could take any of those values, even though the final decision will be made according to our specific needs regarding saving and execution time.

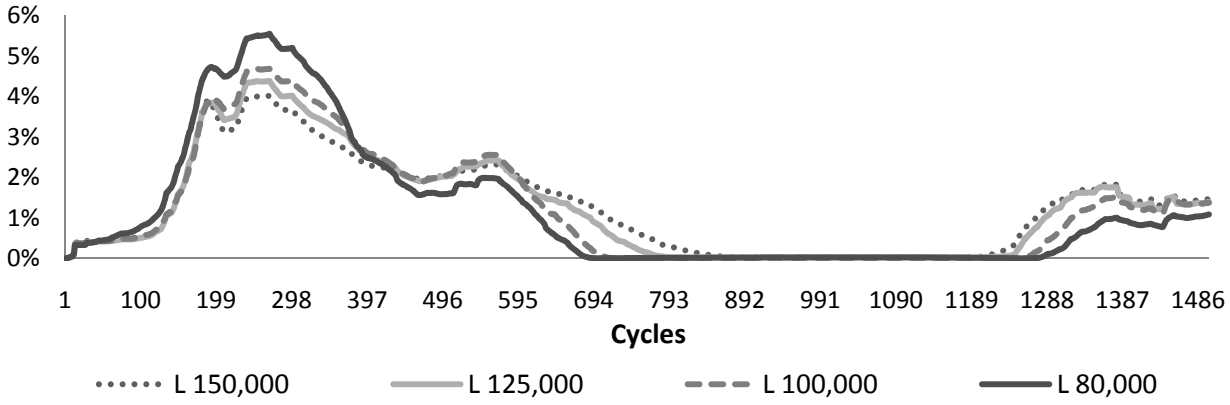


Figure 49: Trade off power saving / execution time [vertical axis = trade off]

6.2.3 Experiment 2: Different number of neighbors

We try varying the number of neighbors with a *high_limit* of 100,000 connections.

In this case is impossible to see any differences in the charts showing the values throughout the day, so we decided to introduce charts with the average values in a day for the different number of links.

In *Figure 44* we see that the power saving does not change much by changing the number of neighbors: 7.86% is the minimum value (when N=2) and 8.98% is the maximum (when N=10). In *Figure 45* we see that the execution time does not change much either, from the maximum value 9.72 cycles (N=10) to 9.31 (N=20). For both cases, the variations are not very important.

Anyway, if we look at *Figure 46*, which shows the relationship between power saving and execution time, we see that from N=7 we have a higher ratio, so it seems a good idea to set this value to at least 7.

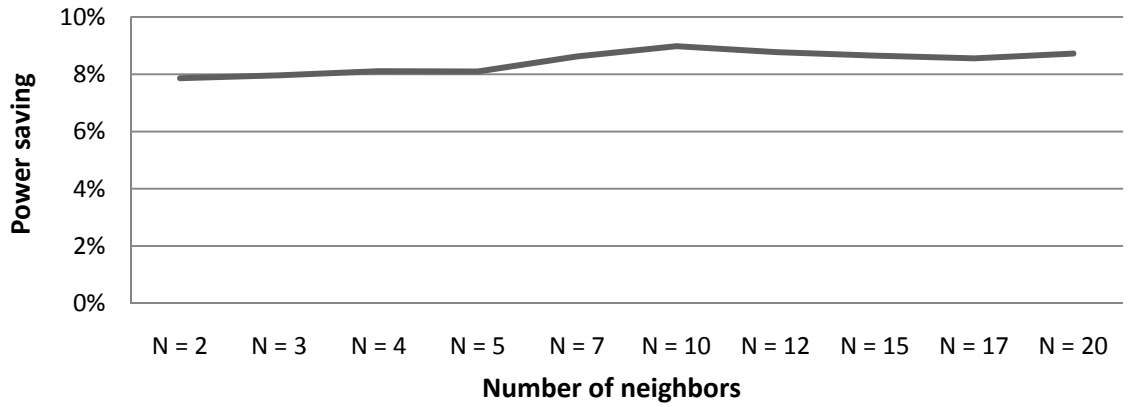


Figure 50: Power saving with different number of links

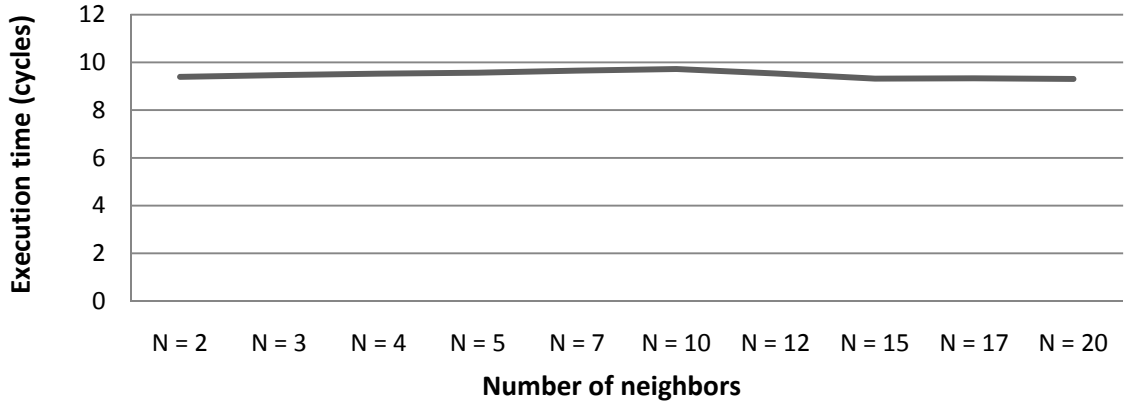


Figure 51: Execution time with different number of links

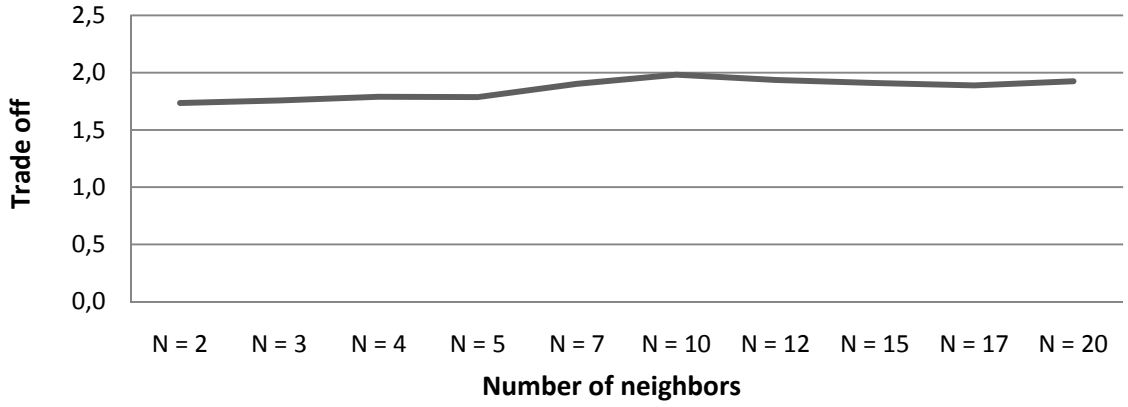


Figure 52: Trade off power saving / execution time

6.3 Load balancing

6.3.1 Scheduled simulations

We have tested our model in two different situations. While in the first one the load remains constant during the whole simulation, in the second one we introduce a load peak.

In both simulations the network has a size of 1,000 nodes: 800 clients and 200 servers. Each server is connected to approximately 20 servers (neighbors) and can process 10,000 tasks per second.

70% of the clients are initially set to low request rate and 30% to medium request rate, which means that 560 clients establish between 10,111 and 11,111 connections to random servers and 240 establish between 21,222 and 22,222 connections.

We test our model with different values for the overload threshold (the number of connections from which a server is considered overloaded): 55,555, 66,666, 77,777, 88,888 and 99,999.

The simulations have a length of 1,000 cycles, even though the results do not show them all since the values reach stability much earlier.

First we test our model in the two traffic situations described before with an available threshold of 66,666 connections (the nodes with a load under that value will try to help their neighbors) and different values for the overload threshold (the number of connections from which a server is considered overloaded): 55,555, 66,666, 77,777, 88,888 and 99,999 connections. This will be *Experiment 1*.

After that, *Experiment* 2 will consist on testing the model with different values for both thresholds (available and overload). We will have 30 simulations for each load situation, with the next values:

- Overload threshold: 55,555, 66,666, 77,777, 88,888 and 99,999 connections.
- Available threshold: 11,111, 22,222, 33,333, 44,444, 55,555 and 66,666 connections.

Finally, in *Experiment* 3 we will analyze the variations in the results when we change the number of neighbors of each node.

6.3.2 Experiment 1: Fixed available threshold (66,666 connections)

6.3.2.1 Constant load

Figure 47 shows the results obtained when the load is constant throughout the simulation.

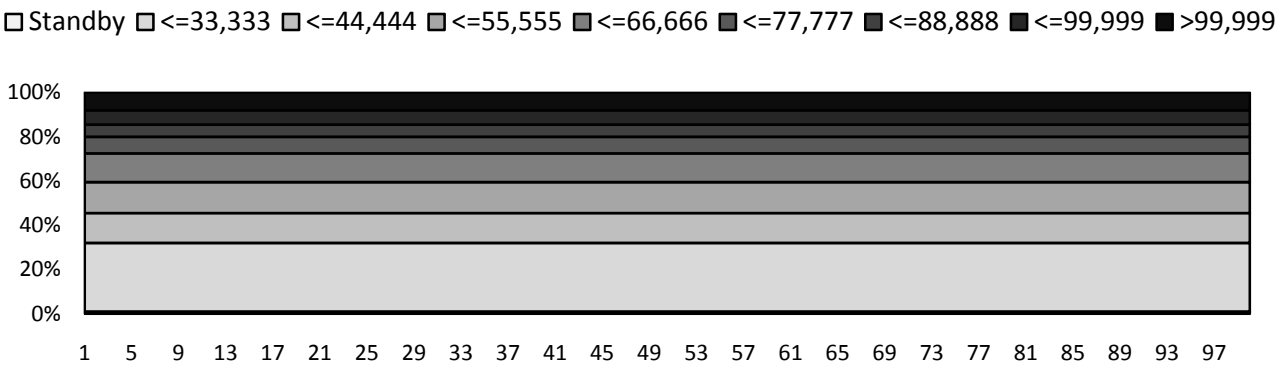


Figure 53a: Percentage of nodes in every state with no algorithms
[vertical axis = nodes in the system, horizontal axis = cycles]

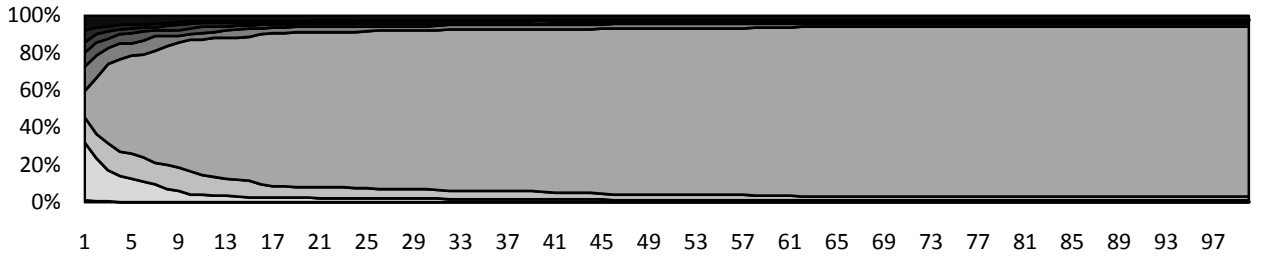


Figure 54b: Percentage of nodes in every state with *overload_threshold* = 55,555
[vertical axis = nodes in the system, horizontal axis = cycles]

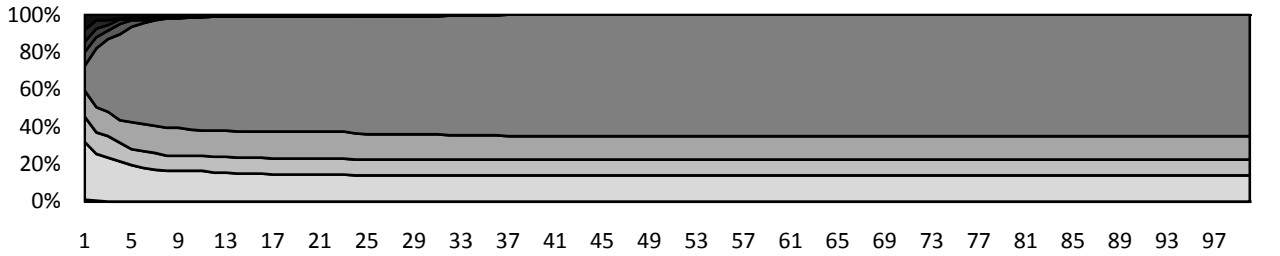


Figure 55c: Percentage of nodes in every state with *overload_threshold* = 66,666
[vertical axis = nodes in the system, horizontal axis = cycles]

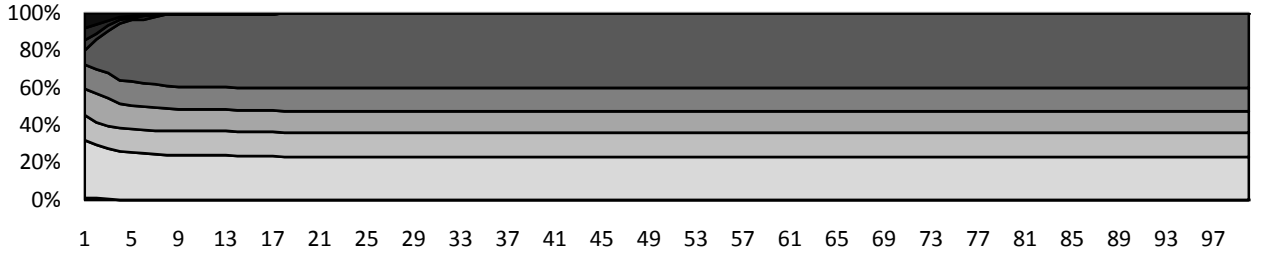


Figure 56d: Percentage of nodes in every state with *overload_threshold* = 77,777
[vertical axis = nodes in the system, horizontal axis = cycles]

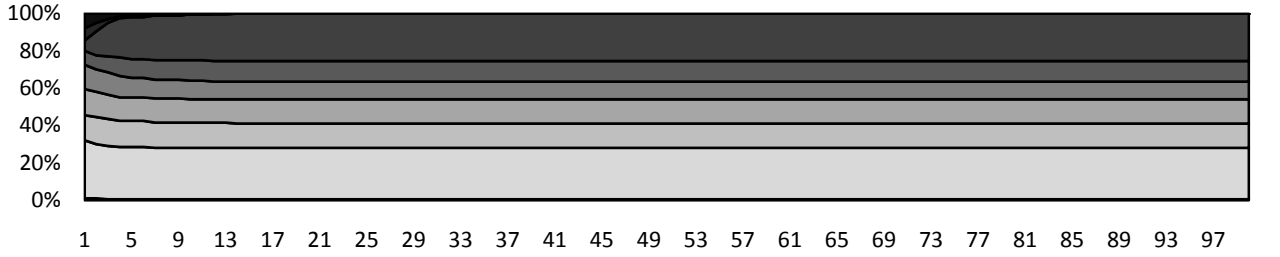


Figure 57e: Percentage of nodes in every state with *overload_threshold* = 88,888
[vertical axis = nodes in the system, horizontal axis = cycles]

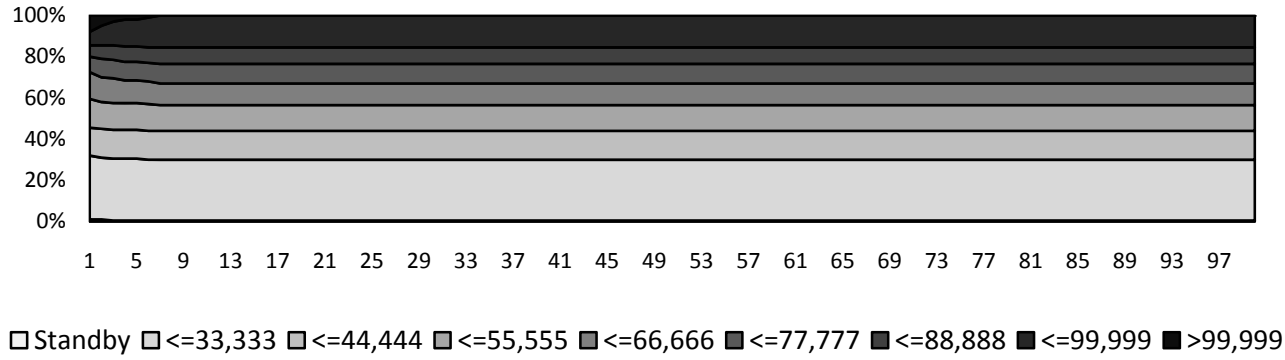


Figure 58f: Percentage of nodes in every state with *overload_threshold* = 99,999
[vertical axis = nodes in the system, horizontal axis = cycles]

Initially we have 1% of the servers in standby, 31% with low load, 40.5% with medium load and 27.5% with high load. After that, it is easy to see that, in every case, load balancing is achieved with the *Overload Algorithm*.

When the overload threshold is 55,555 connections (*Figure 47b*) we see that almost all the nodes with load higher than the threshold disappear, this reducing also the number of low loaded nodes (and of course the few standby nodes we had in the beginning), since the load of higher loaded nodes is distributed among them. It is impossible to distribute the entire load among nodes with a number of connections smaller than 55,555, so we have some remaining high load nodes.

For the rest of cases, the load is distributed with no problems among the nodes, so after some cycles (in any case it is more than 40 cycles) we do not have any nodes with a load over the set threshold.

Besides we see that, from a threshold of 66,666 connections, an important number of low loaded nodes remain after the situation stabilizes, so we can think that introducing an energy saving algorithm could improve the performance of this model.

Smaller values of the *overload threshold* distribute better the load among the servers, so the execution time decreases with that parameter. This can be observed in *Figure 48*.

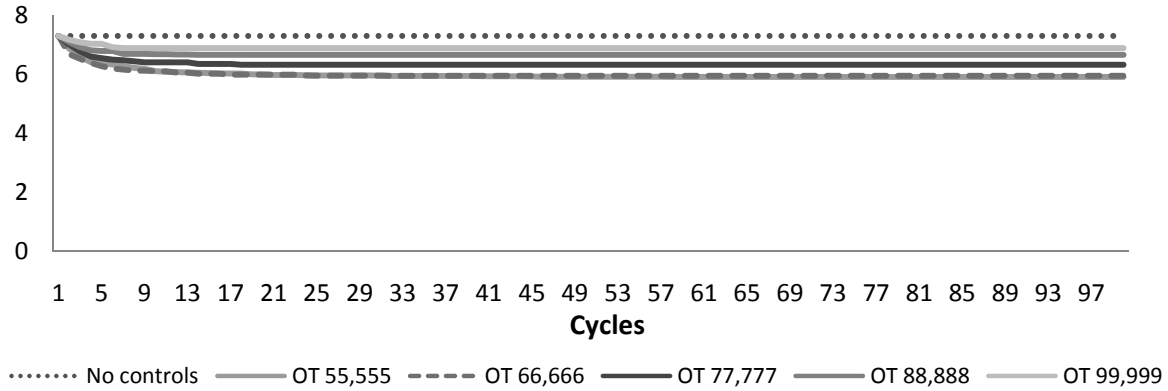


Figure 59: Execution time in the different simulations
[vertical axis = execution time in cycles]

In this case, there is no sense in talking about energy saving, since it is clear that there is no saving here (the savings were substantial only when we were able to put some nodes to sleep).

6.3.2.2 Load peak

With the same initial situation we used before, we introduce a load peak with the class created for this purpose that was explained earlier in this document.

Figure 49 shows the results for this simulation.

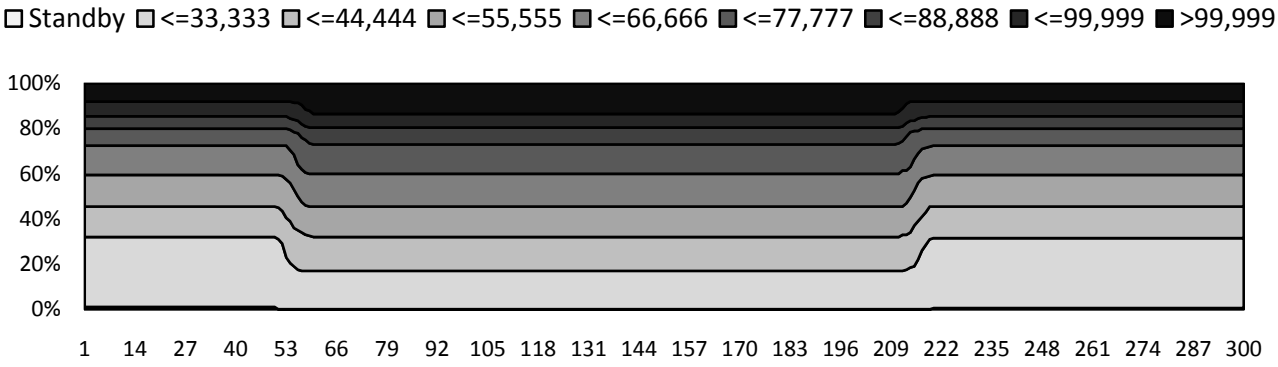


Figure 60a: Percentage of nodes in every state with no algorithms
[vertical axis = nodes in the system, horizontal axis = cycles]

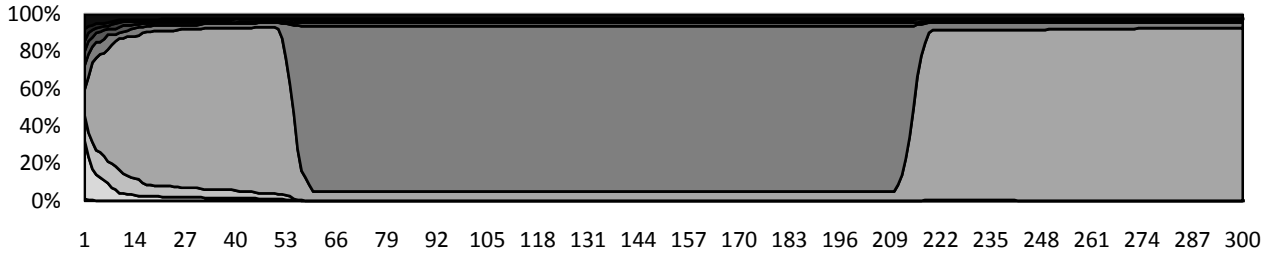


Figure 61b: Percentage of nodes in every state with *overload_threshold* = 55,555
[vertical axis = nodes in the system, horizontal axis = cycles]

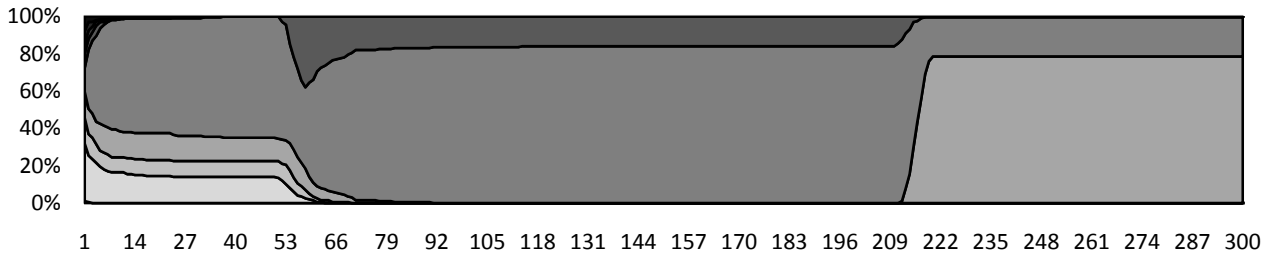


Figure 62c: Percentage of nodes in every state with *overload_threshold* = 66,666
[vertical axis = nodes in the system, horizontal axis = cycles]

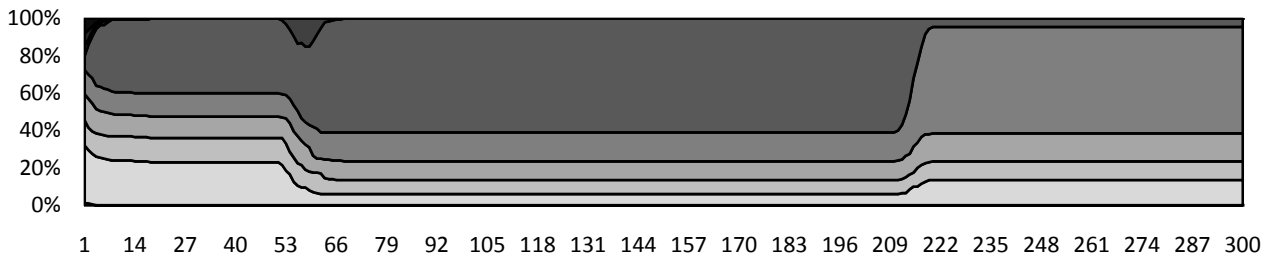


Figure 63d: Percentage of nodes in every state with *overload_threshold* = 77,777
[vertical axis = nodes in the system, horizontal axis = cycles]

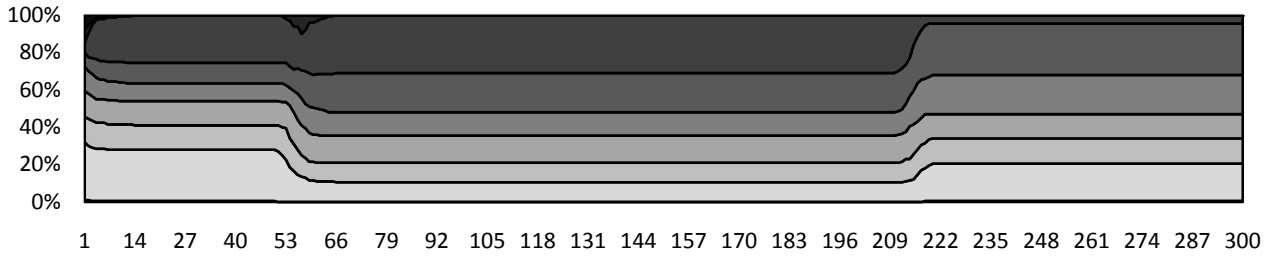


Figure 64e: Percentage of nodes in every state with *overload_threshold* = 88,888 [vertical axis = nodes in the system, horizontal axis = cycles]

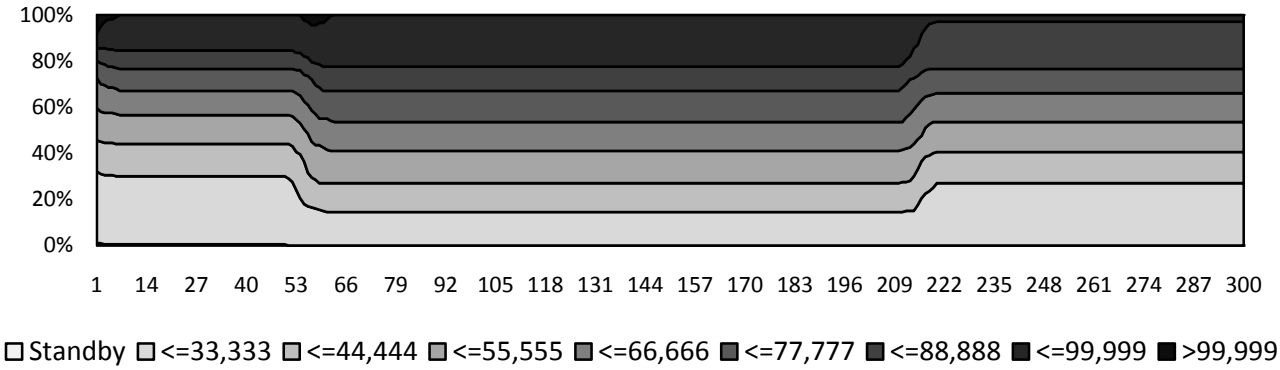


Figure 65f: Percentage of nodes in every state with *overload_threshold* = 99,999 [vertical axis = nodes in the system, horizontal axis = cycles]

We see that, when we do not apply the algorithm, the system starts as in the previous configuration: 1% of the servers in standby, 31% with low load, 40.5% with medium load and 27.5% with high load. After cycle 60 the load is increased, which changes the situation into 17% of the servers with low load, 43% with medium load and 40% with high load. After cycle 220, the load decreases again returning to the initial values.

When we apply the *Overload Algorithm* the results obtained differ depending on the overload threshold.

Again, when the overload threshold is 55,555 connections (*Figure 49b*) not all the nodes with higher load can transfer it, since there are not enough available nodes to take it. For this reason, before the peak starts we have most of the servers with a load between 44,444 and 55,555 connections. During the peak, almost all the nodes have between 55,555 and 66,666 connections. After it, since the same number of connections is removed from each server, we return to the same situation we had before the peak, which is more or less the same we had without the peak.

When the threshold is 66,666 connections (*Figure 49c*) we see that the system manages to distribute the load of all the servers with a load over that value before the peak. However, when the peak arrives there are not enough available nodes to accomplish our objective, so after the system runs out of available nodes, there are still some servers over the threshold. The rest of the nodes have a load between 55,555 and 66,666 connections. After the peak, the same situation we had with the threshold of 55,555 connections occurs, and, even though we do not have any servers over the overload threshold, all the nodes have between 44,444 and 66,666 connections.

The situation changes when the overload threshold is 77,777 connections (*Figure 49d*) or higher. We are able to distribute the load even during the peak and there are still some low loaded nodes in the system. As we stated in the plain load situation, this means that, applying a standby algorithm like we did in previous sections, we could get some energy saving.

Figure 50 shows the execution times for the different values of the *overload threshold*. As happened when there was no peak, the execution time is smaller with smaller values of that

parameter. Nevertheless, we can see that the execution time for a threshold of 55,555 connections is higher than the time for a threshold of 66,666 connections. We find the explanation for this in the fact we explained while analyzing the previous graph. When the threshold is 55,555 connections, not all the load can be distributed, so we still have some high loaded nodes. These nodes increase the execution time. When the threshold is 66,666 connections the entire load can be distributed among the servers, so the execution time is smaller than in the previous case.

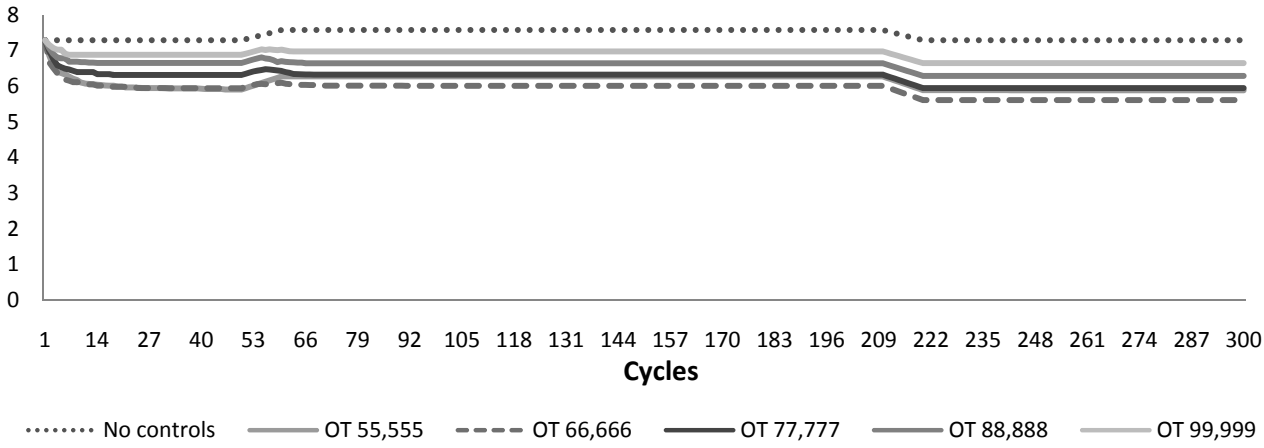


Figure 66: Execution time in the different simulations
[vertical axis = execution time in cycles]

Again, talking about energy saving has no sense, since we do not have any standby nodes.

6.3.3 Experiment 2: Different available threshold

6.3.3.1 Constant load

We want to analyze what happens when we set different values for the available threshold. In this case, since a lot of different simulations have been made, we will just include the charts that summarize the execution time results, since this is what really matters to us in this section.

Figure 51 shows the average values of the execution time in every simulation. We see that for values of the available threshold between 44,444 and 66,666 connections we achieve good execution time values, which mean that the load is well distributed. When the overload threshold is 55,555 connections the execution time should be under 5.5 if the load was completely distributed, but we saw in *6.3.2* that it is not possible to distribute the entire load among nodes with a number of connections under 55,555.

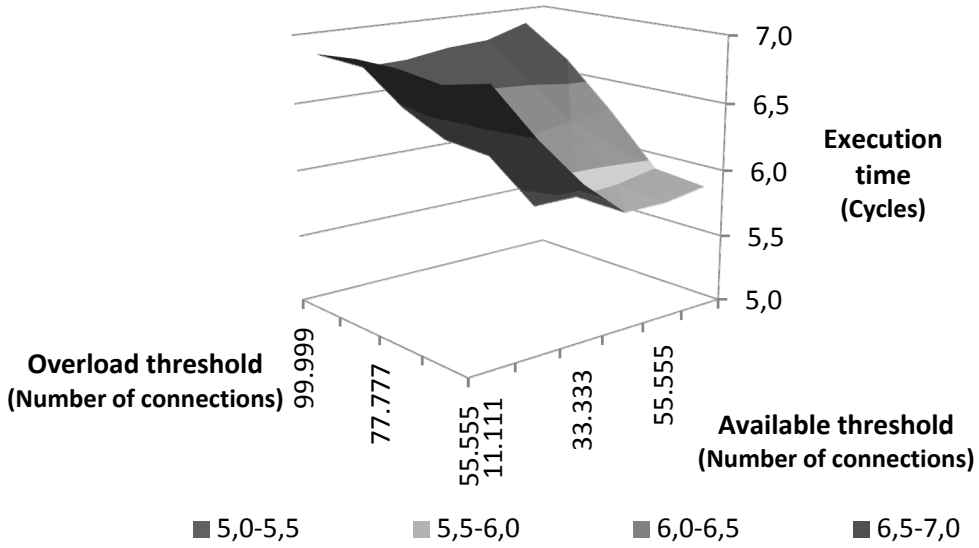


Figure 67: Average execution time in the different simulations

For values of the available threshold of 33,333 connections and smaller we see that the execution time increases. This is obvious, since reducing this threshold means also reducing the availability of the nodes.

6.3.3.2 Load peak

Figure 52 shows the average execution time obtained when we introduce a load peak in the simulation. As happened when we had constant load (*Figure 51*), the execution time is low for an available threshold of 44,444 connections or higher and starts to increase as we reduce this value.

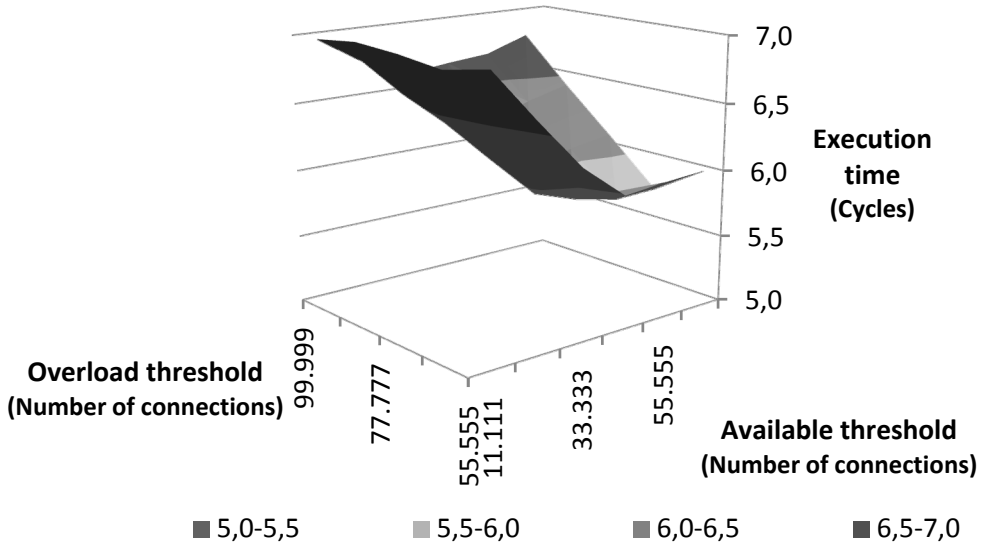


Figure 68: Average execution time in the different simulations

In light of these results, we should set an available threshold of at least 44,444 connections if we want a good distribution of the load in the system.

6.3.4 Experiment 3: Different number of neighbors

Again we present the charts of the average values obtained in every case.

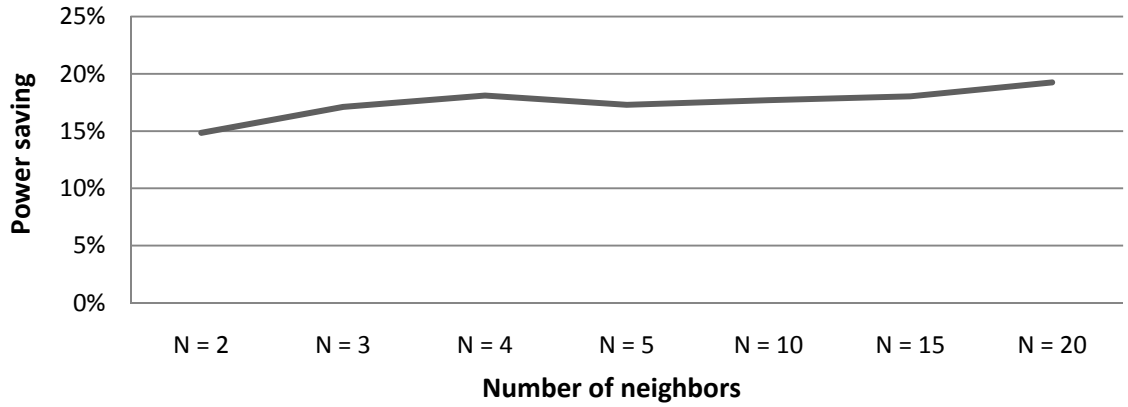


Figure 69: Power saving with different number of links

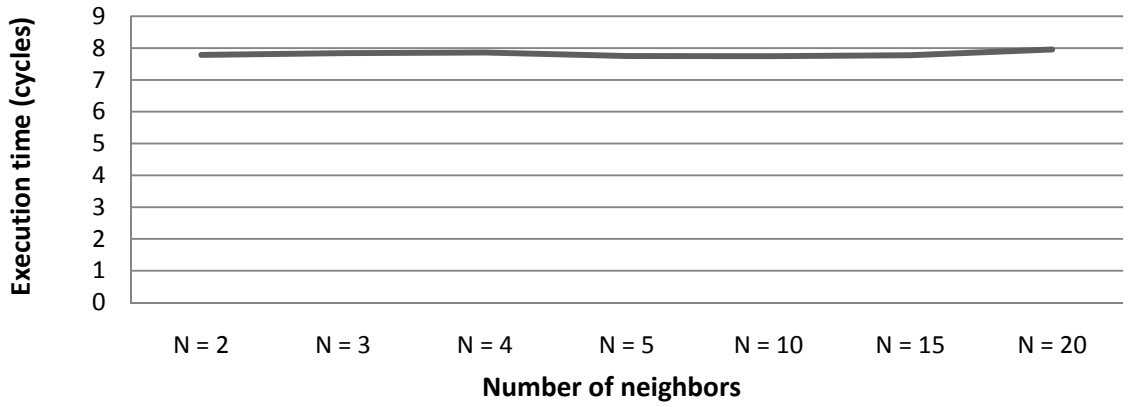


Figure 70: Execution time with different number of links

In *Figure 53* we see that, as we increase the number of links, the average power saving also increases, passing from 14.84% (when N=2) to 19.25% (when N=20).

It seems that N = 2 is the only value we should avoid, since for the rest of values assigned to that parameter the average saving obtained is over 17%.

The execution time (*Figure 54*) presents small variations, being always inside the interval 7.75-7.95. If we are not extremely sensitive about this value, we will base our decision on the saving we aim to achieve.

6.4 Load balancing and standby

6.4.1 Scheduled simulations

Again, we test our model in two different situations. While in the first one the load remains constant during the whole simulation (*6.4.2*), in the second one we introduce a load peak (*6.4.3*).

In both simulations the network has a size of 1,000 nodes: 800 clients and 200 servers. Each server is connected to approximately 20 servers (neighbors) and their process capability is 10,000 tasks.

70% of the clients are initially set to low and 30% to medium, which means that 560 clients establish between 10,111 and 11,111 connections to random servers and 240 establish between 21,222 and 22,222 connections.

The simulations have a length of 1,000 cycles, even though the results do not show them all since the values reach stability much earlier.

In the two different load situations, our model is tested in 30 different cases, where the values for the different thresholds are:

- Overload threshold: 55,555, 66,666, 77,777, 88,888 and 99,999 connections.
- Available threshold: 11,111, 22,222, 33,333, 44,444, 55,555 and 66,666 connections.

6.4.2 Experiment 1: Constant load

Since the number of cases analyzed in this section is too big, we did not include the charts corresponding to every experiment but some summarizing the results obtained.

Figure 55 shows the average power saving achieved in the different simulations. We observe that the higher the thresholds are the higher the saving gets. As usually, high savings imply high execution times, as can be seen on *Figure 56*.

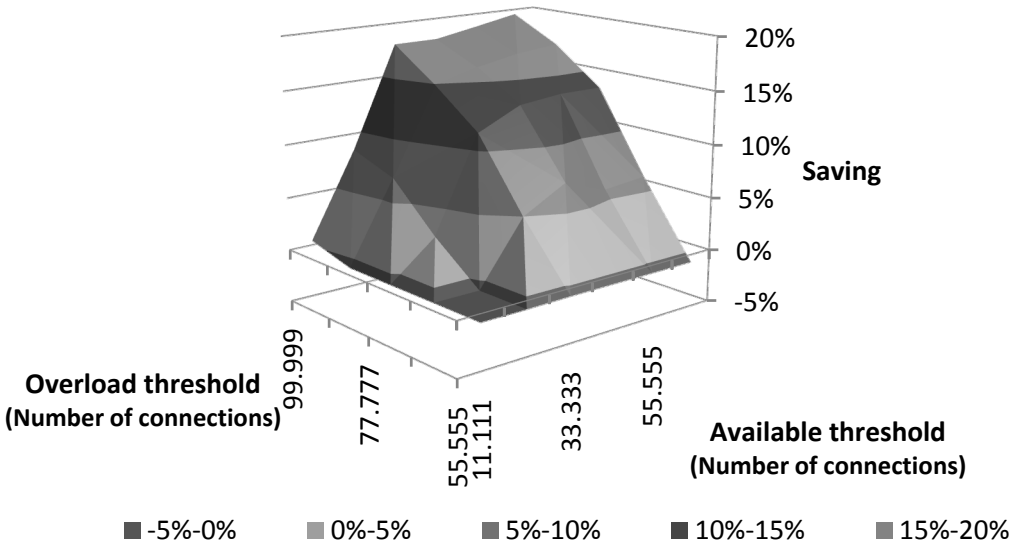


Figure 71: Average energy saving in the different simulations

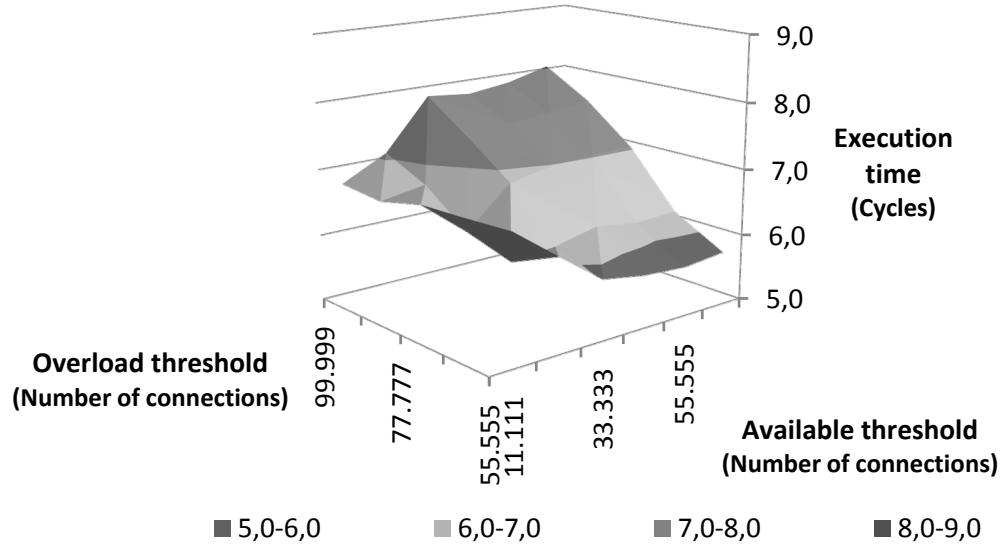


Figure 72: Average execution time in the different simulations

Finally, *Figure 57* shows the relationship between power saving and execution time. We see that the maximum value is achieved when both thresholds take their maximum value. Anyway, we will not usually be able to reach this value, since we have to base our decision on our needs regarding execution time and energy saving.

From this figure we can also see that the overload threshold should be in any case bigger than 55,555 connections if we want to get any savings. Something similar happens with the available threshold, which should be at least 22,222 connections to achieve some savings.

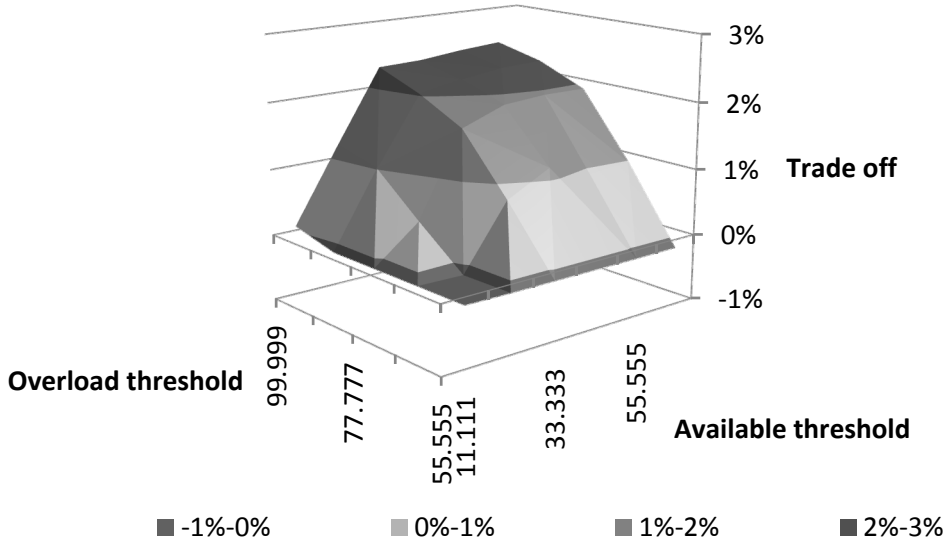


Figure 73: Trade-off power saving / execution time

6.4.3 Experiment 2: Load peak

Obviously, when we introduce the load peak the power saving is reduced. This can be seen on *Figure 58*.

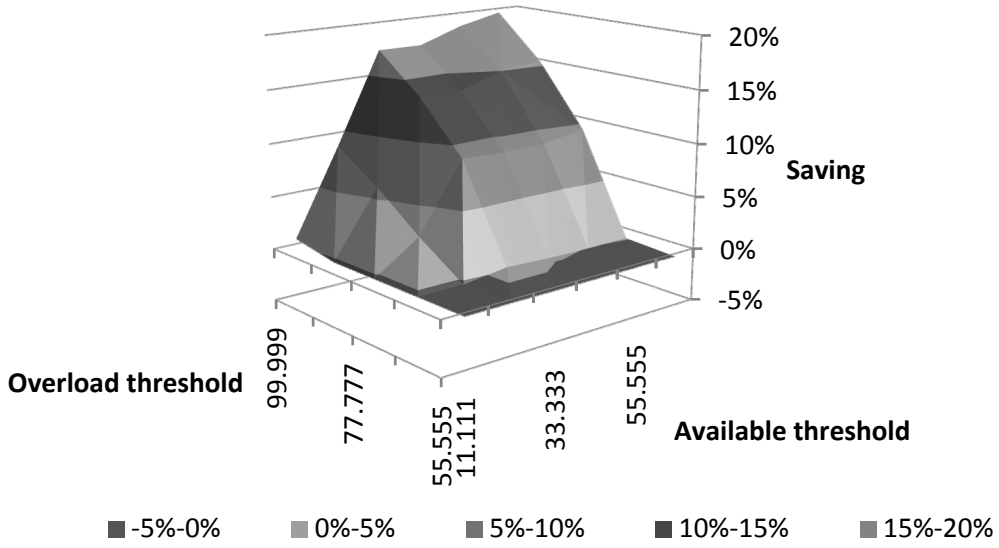


Figure 74: Average energy saving in the different simulations

As in the previous situation, *Figure 59* shows the average execution time and *Figure 60* the trade-off between power saving and execution time.

Even though we have smaller saving values, we are in the same situation as before: we get the bigger savings with the higher threshold values, but that increases also the execution time. We will have to choose those values according to our needs.

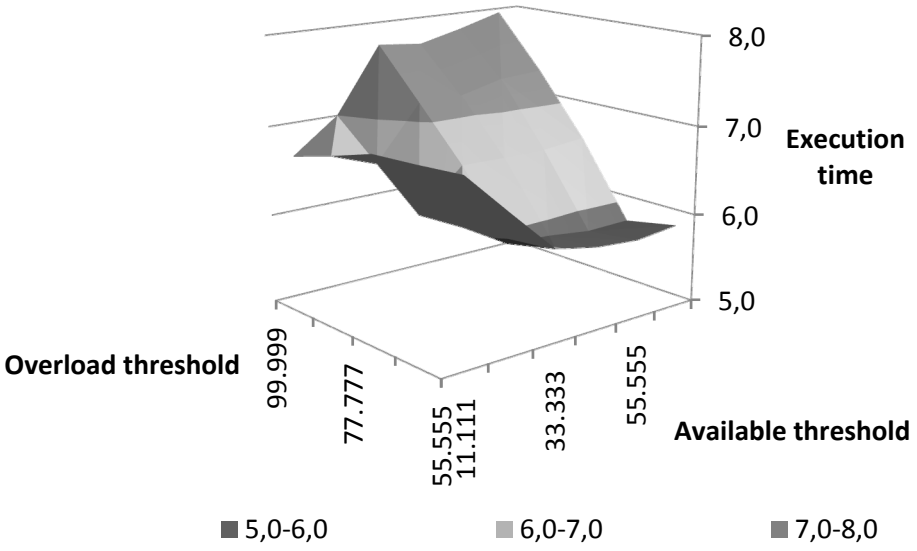


Figure 75: Average execution time in the different simulations

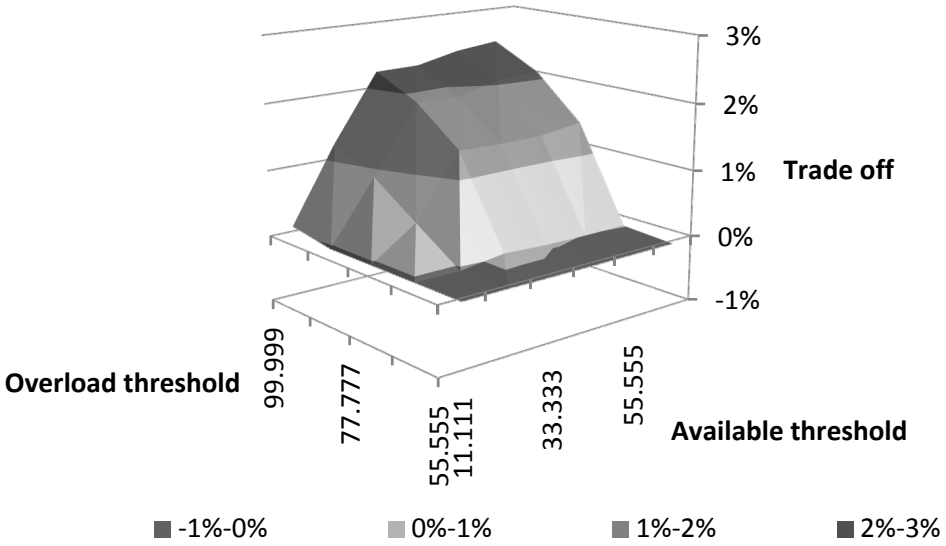


Figure 76: Trade-off power saving / execution time

Again, the minimum value of the available threshold to be able to save some energy should be 22,222 connections. The overload threshold should be higher in this case than before, since with a value of 66,666 connections we can save less than 0.3% energy. We will set it to, at least, 77,777 connections.

7. Analysis of the results

Through *Section 6* we have seen the performance of the different models simulated, depending on the variation of a set of parameters. We can basically distinguish two scenarios: distributed server farm and client-server model.

In those two scenarios we repeat the same process: first we evaluate the behavior of the system with a flat load level and, after that, we analyze the performance of the system with a more realistic load shape. In the case of the distributed server farm, this realistic traffic corresponds to the one presented in [24]. In the client-server scenario we do not use such a realistic traffic model; we just increase the load in the system for about 170 simulation cycles. This difference between the two models explains why, in the first case, the reduction on energy saving is far more important than in the second case, where during most of the simulation we still have a flat traffic shape.

However, in both cases we see that important saving values are achievable (around 10% and over) with the application of the simple algorithms described. As we said through *Section 6*, the higher the saving is the higher the execution time gets, so in any case the administrator of the system will have to set the parameters of the model according to the performance expected.

7.1 Comparison with respect to other solutions

In *Section 3.4* we reviewed some of the work related to the same topic studied in this project. Different approaches were presented, but we did not find a piece of work that we could compare quantitatively to our work, so the only comparisons we can establish in this section are of qualitative nature.

In most of the papers (like [24, 28, 29, 30]) the algorithms used are far more complex than the ones used in our work and take into account many variables. This also represents a problem when trying to compare those solutions to ours, which, in fact, is rather simple.

The main differences between some of the works mentioned in *Section 3.4* and our project are the following:

- On [24], a lot of simulations and results on load balancing and the impact on power saving (with saving values up to 30.8%) are presented. However, the approach is currently not autonomic and decentralized as it is proposed in our project, so we cannot compare the numeric values obtained in that paper to the ones obtained in our simulations.
- Unlike the approach described in [29], ours is a distributed one: each node can carry on the power saving algorithm; and other parameters are taken in consideration, like low, medium and high load, not only idle or busy state.
- The work in [30] presents a power saving of approximately 10%. Nevertheless, the approach presented is not autonomic, since it is based on the presence of two managers that have a central control on the performance of the servers.
- Even though the results on [33] are interesting, the aspect of power saving is not currently covered, so it is also difficult to compare it to our work, where power saving is one of the main aspects.
- Another load balancing approach is showed on [34], but the presence of the dispatcher contrasts again with the idea of autonomic computing presented in our project. Besides,

that paper does not consider the matter of energy saving either (it just focuses on execution time).

8. Conclusions and future work

The work presented in this document shows that it is possible to achieve important saving values in distributed data centers, while preserving reasonable execution times, by the introduction of simple algorithms that are executed at a local level.

On the first of the proposed scenarios (distributed server farm) we achieve a maximum power saving of 11.66%, comparing to the power consumed before applying the algorithms. On the second one (client-server), the maximum achieved saving rises up to 19.56%, thanks to an enhancement of the algorithms used in the first case.

Besides this, on the second scenario we also manage to reduce the average execution time in the system. This is possible only in the client-server model and not in the distributed server farm because in the latter the load is modeled to be completely balanced in the beginning, so it's impossible to get to a better situation. On the contrary, on the client-server model the initial connections are randomly established between clients and servers, so the load is not balanced, i.e. not optimally distributed among the servers.

However, as we have stated previously in some sections, this project has to be considered as a preliminary work, since it is just a theoretical approach for solving the problem presented in the beginning. With this as a starting point, a lot of work has yet to be done in this field to transfer the theoretical analysis to real server farms.

Some of the points that will have to be taking into account are:

- *Realism in the simulations*: before translating this model into a real implementation, a first step could be enhancing the simulated model by adding more realism to it. In the systems we simulated in this document we did not include dynamism: the number of nodes present in the system remains constant during all the simulation and the possibility of a node crashing and leaving the system is not considered. Besides these concepts, we could take into account other values such as the time needed to shut down a server or start it up again or the existence of different kind of servers able to execute different services.
- *Self-organization for node clustering*: considering the distributed server farm as composed by different kinds of nodes, self-organization could be applied for optimizing the overlay network by interconnecting servers with similar functional characteristics (e.g., able to execute the same services).
- *Optimization based on location*: besides the consideration regarding the type of the servers, the overlay could also be optimized by considering their location (e.g., select connections between near servers).
- Implementation of the logic according to the ACE toolkit embedded supervision approach presented in CASCADAS.

9. References

[1] U.S. Environmental Protection Agency ENERGY STAR Program. *Report to Congress on Server and Data Center Energy Efficiency Public Law 109-431.*
[2] IST Project CASCADAS Deliverable 3.1
[3] Jeffrey O. Kephart and David M. Chess. *The Vision of Autonomic Computing*. Computer, Vol. 36, No. 1. (January 2003), pp. 41-50.
[4] *Component-ware for Autonomic Situation-aware Communications, and Dynamically Adaptable Services (CASCADAS)*. www.cascadas-project.org
[5] E Bonabeau, M Dorigo and G Theraulaz. *Swarm Intelligence: From Natural to Artificial Systems.* Oxford University Press, New York, 1999.
[6] IST Project CASCADAS Deliverable 3.7
[7] P Marrow. *Nature-inspired computing technology and applications.* BT Technology Journal Vol. 18, No. 4, 2000, pp. 13-23.
[8] S Bocaletti, V Latora, Y Moreno, M Chavez and D-U Hwang. *Complex networks: structure and dynamics.* Physics Reports, Vol. 424, 2006, pp. 175-308.
[9] S Milgram. *The small world problem.* Psychol. Today, Vol. 2, 1967, pp. 60-67.
[10] D J Watts and S H Strogatz. *Collective Dynamics of 'small world' networks.* Nature, Vol. 393, 1998, pp. 440-442.
[11] D J Watts. *Small Worlds.* Princeton University Press, Princeton, 1999.
[12] S H Strogatz. *Exploring complex networks.* Nature, Vol. 410, 2001, pp. 268-276.
[13] M Faloutsos, P Faloutsos and C Faloutsos. *On power-law relationships of the internet topology.* Comp. Comm. Rev., Vol. 29, 1999, pp. 251-262.
[14] P Erdos and A Renyi. *On the evolution of random graphs.* Publ. Math. Inst. Hung. Acad. Sci., Vol. 5, 1960, pp. 17-61.
[15] M E J Newman. *The structure and function of complex networks.* SIAM Review Vol. 45, No. 2, 2003, pp. 167-256.
[16] O Babaoglu, H Meling and A Montressor. *Anthill: A framework for the development of agent-based peer-topeer systems.* Proc. Int. Conf. Distributed Computer Systems 2002 (ICDCS 2002), 2002.
[17] O Babaoglu, G Canright, A Deutsch, G Di Caro, F Ducatelle, L Gambardella, N Ganguly, M Jelasity, R Montemanni and A Montressor. *Design patterns from biology for distributed computing.* Proc. European Conference on Complex Systems, Paris, 2005.
[18] C Bernon, V Chevrier, V Hilaire and P Marrow. *Applications of self-organising multi-agent systems: an initial framework for comparison.* Informatica, Vol. 30, 2006, pp. 73-82.
[19] G Di Marzo Serugendo, N Foukia, S Hassas, A Karageorgos, S K Mostéfaoui, O F Rana, M Ulieru, P Valckenaers and C Van Aart. *Self-organisation: paradigms and applications, Engineering Self-Organising Systems: nature-inspired approaches to software engineering.* LNAI 2977, Springer, Berlin, 2004, pp. 1-19.
[20] P E Turner and L Chao. *Escape from Prisoner's Dilemma in RNA phage Φ6.* Am. Nat., Vol. 161, 2003, pp. 497-505.
[21] J von Neumann and O Morgenstern. *The Theory of Games and Economic Behavior.* Princeton University Press, Princeton, 1944.
[22] Fabrice Saffre, Richard Tateson, José Halloy, Mark Shackleton and Jean Louis Deneubourg *Aggregation Dynamics in Overlay Networks and Their Implications for Self-Organized Distributed Applications*.

[23] Márk Jelasity, Alberto Montresor and Ozalp Babaoglu. *Gossip-Based Aggregation in Large Dynamic Networks.* ACM Trans. Comput. Syst., 23(1):219-252, August 2005.
[24] Gong Chen, Wenbo He, Jie Liu, Suman Nath, Leonidas Rigas, Lin Xiao and Feng Zhao. *Energy-Aware Server Provisioning and Load Dispatching for Connection-Intensive Internet Services.* NSDI 2008.
[25] Balasubramanian Seshasayee, Ripal Nathuji and Karsten Schwan. *Energy-Aware Mobile Service Overlays: Cooperative Dynamic Power Management in Distributed Mobile Systems.* Fourth International Conference on Autonomic Computing, 2007. ICAC '07.
[26] Luciano Bononi, Marco Conti and Lorenzo Donatiello. *A Distributed Mechanism for Power Saving in IEEE 802.11 Wireless LANs*. Mobile Networks and Applications, Volume 6, Number 3, June 2001 , pp. 211-222(12).
[27] Benjie Chen, Kyle Jamieson, Hari Balakrishnan and Robert Morris. *Span: An Energy Efficient Coordination Algorithm for Topology Maintenance in Ad Hoc Wireless Networks*. 7th ACM International Conference on Mobile Computing and Networking (MobiCom '01), Rome, Italy, July 2001.
[28] Dara Kusic, Jeffrey O. Kephart, James E. Hanson, Nagarajan Kandasamy and Guofei Jiang. *Power and Performance Management of Virtualized Computing Environments via Lookahead Control.* Fifth International Conference on Autonomic Computing, 2008. ICAC '08.
[29] Yung-Hsiang Lu, Eui-Young Chung, Tajana Šimunić, Luca Benini and Giovanni De Micheli. *Quantitative Comparison of Power Management Algorithms*. Design Automation and Test in Europe, Stanford University, 20-26, March, 2000.
[30] Jeffrey O. Kephart, Hoi Chan, Rajarshi Das, David W. Levine, Gerald Tesauro, Freeman Rawson and Charles Lefurgy. *Coordinating Multiple Autonomic Managers to Achieve Specified Power-Performance Tradeoffs*. Fourth International Conference on Autonomic Computing, 2007. ICAC '07. [31] George Cybenko. *Dynamic Load Balancing for Distributed Memory Multiprocessors*. Journal of Parallel and Distributed Computing, v.7 n.2, p.279-301, October 1989.
[32] Jacques Bahi, Raphael Couturier and Flavien Vernier. *Synchronous Distributed Load Balancing on Dynamic Networks*. Journal of Parallel and Distributed Computing, v.65 n.11, p.1397-1405, November 2005.
[33] Elisabetta Di Nitto, Daniel Dubois, Raffaela Mirandola, Fabrice Saffre and Richard Tateson. *Self-Aggregation Techniques for Load Balancing in Distributed Systems*. Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems, 2008. SASO '08.
[34] David Breitgand, Rami Cohen, Amir Nahir and Danny Raz. Fourth International Conference on Autonomic Computing, 2007. ICAC '07.
[35] http://peersim.sourceforge.net
[36] Salvador Sahuquillo, Antonio Manzalini, Corrado Moiso, Josep Solé-Pareta, Beatriz Otero. *Power saving in Data Centers: A use case for self-organization*. CASCADAS Project Demonstration Event. London, October 2008.