A FAULT TOLERANT, PEER-TO-PEER BASED SCHEDULER FOR HOME GRIDS

A thesis submitted to the
University of Kent in the subject
of Computer Science for the
degree of Doctor of Philosophy

By
Erick Lopes da Silva
January 2012

Abstract

This thesis presents a fault-tolerant, Peer-to-Peer (P2P) based grid scheduling system for highly dynamic and highly heterogeneous environments, such as home networks, where we can find a variety of devices (laptops, PCs, game consoles, etc.) and networks.

The number of devices found in a house that are capable of processing data has been increasing in the last few years. However, being able to process data does not mean that these devices are powerful, and, in a home environment, there will be a demand for some applications that need significant computing resources, beyond the capabilities of a single domestic device, such as a set top box (examples of such applications are TV recommender systems, image processing and photo indexing systems). A computational grid is a possible solution for this problem, but the constrained environment in the home makes it difficult to use conventional grid scheduling technologies, which demand a powerful infrastructure.

Our solution is based on the distribution of the matchmaking task among providers, leaving the final allocation decision to a central scheduler that can be running on a limited device without a big loss in performance.

We evaluate our solution by simulating different scenarios and configurations against the Opportunistic Load Balance (OLB) scheduling heuristic, which we found to be the best option for home grids from the existing solutions that we analysed. The results have shown that our solution performs similar or better to OLB. Furthermore, our solution also provides fault tolerance, which is not achieved with OLB, and we have formally verified the behaviour our solution against two cases of network partition failure.

Contents

List of Publications

1. Silva, E. L., Linington, P., *A P2P Based Scheduler for Home Grids*, RECENT TRENDS IN WIRELESS AND MOBILE NETWORKS, Communications in Computer and Information Science, 2011, Volume 162, Part 3, 325-336, Springer (http://www.springerlink.com/content/h3nx27221m5n628m/). June 2011.

Table of Figures

Table of Tables

Acknowledgement

It is the end of another tough journey, and I would like to thank the people that made this work possible.

I am very grateful to my supervisor, Professor Peter Linington, for his constructive criticism and guidance during the period of this work. His experience was very important to improve style and correctness in the thesis.

I also owe my gratitude to the late Dr. David Shrimpton for his support and dedication during the first year of my PhD.

I would also like to thank Professor Michael Kölling and Mr. Ian Utting for the comments and suggestions for the improvement of this work during the panel meetings.

A special thank to Dr. Rodolfo Gomez for being a great friend during all these years of my PhD and for helping me with the model checking presented in the thesis.

Many thanks to my great friend Natália for being always present and supportive whenever I needed. Thanks also to Susanne, Gift, Ismini, Valeschka, Paulo, Olga, Carlos and Márjory for being like a family to me and make this journey easier.

Thanks to the Programme Alban and to the School of Computing of the University of Kent for the financial support for this project.

Many thanks to my friend Corine for helping me to be awake during the nights that I had to work and for the amazing chocolates that were sent to me. Additionally, thanks to my dear friends that were always present: Michèle, Ana, Julie, Lauryn, Fábio, Bárbara, Yvonne and Sílvia.

A very special thanks to Dr. Marcília Campos for always believing in me since my undergraduate degree.

Finally, I cannot find words to describe how grateful I am to my parents, Ednilson and Sebastiana, and to all my family, especially to my brother Eberth and my sister Erika, for the unconditional love, support and encouragement that they have given to me.

To my parents Ednilson and Sebastiana.

"Start by doing what is necessary, then what is possible, and suddenly you are doing the impossible."

(St. Francis of Assisi)

Chapter 1. Introduction

1 Introduction

The number of devices found in a house that are capable of processing data has been increasing in the last few years. However, being able to process data does not mean that these devices are powerful, and they may not be able to run complex applications within an acceptable time for the user.

One example of this kind of application is a recommender system [1-4] for digital television, where the user provides their preferences about TV shows and/or movies, and the system provides him with some suggestions of what to watch. To identify suitable shows or movies, the system usually compares the preferences of the user with the preference of other users with similar tastes, by applying some matching techniques to the recorded data. The techniques that provide the best results for the recommendation (e.g. the use of Single Value Decomposition – SVD [5-8]) usually demand much processing power, and for that reason they cannot be implemented on Set-Top Boxes (STBs) that are usually limited in terms of their processing capabilities. More information on recommender systems is presented in Appendix A of this thesis.

Distributing the processing of an application to more powerful devices is one alternative for extending the capabilities of the limited components and so allowing them to run complex applications. However, it is necessary to define, for example, the mechanism to find the powerful devices that are available, how the devices should communicate and how to decide what devices should receive which processes.

A Computational Grid [9-13] emerges as a suitable option for these challenges, since it has been designed for the distribution of application processing in dynamic and heterogeneous environments, which happens to be the case for home networks.

The following section describes the assumptions and the objectives of this thesis.

1.1 Research Assumptions and Focus

This thesis is concerned with the architecture for provision of home grids. It assumes that:

a) in a home environment there will be a demand for some applications that need significant computing resources, beyond the capabilities of a single domestic device, such as a set top box (examples of such applications are TV recommender systems, image processing and photo indexing systems);
b) there will be a significant number of devices available, which can be used in combination to fulfil the demand;
c) many of the devices will be mobile, or at least portable, so that they may be removed from, or returned to, the home without warning;
d) constraints on the home's energy footprint will result in these devices being switched on and off in an unpredictable way; some devices will be in continuous operation (e.g. associated with domestic services like refrigeration or security) but economy will dictate that these have quite limited capabilities;
e) the devices will have different hardware architectures and capabilities, so that not all devices will be suitable for the support of all the required applications;
f) the devices connect to each other through a router (wirelessly or using cable); some devices may not be able to connect directly with the router (because they use a technology not supported by the router) and will take part in the network using another device as a proxy;
g) the main objective is to perform application tasks as quickly as possible, and it is assumed that the tasks are free-standing, computationally intensive pieces of work;
h) the grid system may receive tasks from different applications simultaneously.

The aim of this thesis is to propose an architecture and supporting protocols to manage the available resources. It describes:

a) the registration and identification of devices;
b) the discovery of suitable resources;
c) the negotiation of constraints on their use (matchmaking);
d) the scheduling and allocation of tasks on devices;

e) fault tolerant mechanisms for the system, including election and recovery of a new scheduler after failure.

The resources are not available to investigate the performance and efficiency of the proposed architecture in a wide variety of configurations, and so the main thrust of the investigation will be by discrete event simulation, supported by benchmarking to characterize available devices and validate the simulation.

1.2 Outline of the thesis

This thesis is organized using the following structure:

- **Chapter 2**: this chapter presents the types of applications that are expected to use our proposed system, and it also shows the possible devices and network set-ups for connecting devices in the home environment;
- **Chapter 3**: presents the literature review for the thesis, including an overview of grid systems, the state of the art of scheduling mechanisms and grids for limited devices;
- **Chapter 4**: describes the architecture and protocols proposed in this thesis;
- **Chapter 5**: describes the methodology for evaluation of the proposed system and the simulation models;
- **Chapter 6**: describes the simulated scenarios and analyze their respective results;
- **Chapter 7**: presents the formal verification of the proposed system for two cases of network partition;
- **Chapter 8**: presents the final considerations and possible future work.

At the end, we present the bibliography used for this thesis and the supporting material as appendixes.

Chapter 2. The Home Environment

2.1 Applications

The solution presented in this thesis was designed with the main objective of creating an infrastructure that enhances the computational capabilities of the devices in the home network and allows more complex applications to be executed faster.

For example, recommender systems usually provide recommendations for a single person using the preferences from other people with similar tastes. As mentioned in the Chapter 1, to achieve better accuracy, the recommender systems must rely on techniques that demand high computational power such as SVD or genetic algorithms. In this case, one can argue that the STB can compute the recommendations in an offline mode, before the user needs it (e.g. while the user sleeps), and give a quick response when asked for it, which is a valid approach.

However, when we consider a novel recommender system that provides recommendations to a group of people, it may be required that the recommendations be computed on demand, in the situations where the data from all the participants cannot be known *a priori* for offline processing. In this scenario, a STB on its own could take a long time to produce an accurate result. This computation could be done more quickly if the STB could use the computational power of other devices in the home (through a domestic computational grid, for example).

Another example of an application that could use the grid infrastructure is a video surveillance system where images captured by cameras are analysed in a search for criminals. Such application could use the computational power of the grid systems running in the homes in the neighbourhood. Facial recognition algorithms are computationally expensive, which makes them appropriate to run in a grid system. For example, research like that presented in [14] and [15] proposes the deployment of face recognition algorithms in a grid system.

Further examples of applications include health systems. With the aging of the population, this kind of application will become more common, and they will become an economically viable solution for patient care [16]. In-home and nursing-home pervasive networks may assist residents and their caregivers by providing continuous medical monitoring (e.g., some of them are devoted to continuous medical monitoring for degenerative diseases like Alzheimer's, Parkinson's or

similar cognitive disorders), memory enhancement, control of home appliances, medical data access, and emergency communication [17].

Some of these applications will be realised by using pervasive wearable sensors that can send the collected data to a central device, which can then use the grid to produce result in a shorter time. The work presented in [18] uses a grid to reduce the total execution time of an electrocardiogram application that analyses complicated signals. With the improvements of digital stethoscopes, we can also envisage computational grids being used to improve the performance of applications that analyse the signals captured by those devices.

The applications mentioned here may involve high network usage to transfer the matrices of preferences, in the case of the recommender system, or to transmit the images captured by the cameras, in the case of the surveillance system, for example. For that reason, grid systems for the home environment should attempt to minimize the use of the network by their own protocols.

A home grid solution can also serve as a resource for conventional grid applications such as SETI@Home[1] and FightAIDS@Home[2], where one of the devices connected to the Internet, for example, could distribute the tasks to the home grid.

In the following sub-sections, we describe the requirements from some of these applications on the grid.

2.1.1 Face Recognition

Face recognition techniques are very useful for a number of applications that can be used in the home environment such as surveillance systems, identity authentication, access control to the property, content-based indexing and video-retrieval systems, for example.

It is well known that it gets harder to distinguish faces as the database grows, since growth reduces the variances between the faces, affecting the recognition accuracy of the system [19].

In order to solve this problem, Zhang et. Al. [19] proposes dividing the large database into sub-databases by maximizing the variance in the sub-databases, but reducing the variance between them.

---

[1] SETI@Home website: http://setiathome.berkeley.edu/
[2] FightAIDS@Home website: http://fightaidsathome.scripps.edu/

In order to improve performance, the solution in [19] keeps copies of the database in the provider nodes with enough storage capabilities in the grid, so that they only need to distribute the images that will have to be analysed and distribute the recognition processes that will use the sub-databases.

Since the recognition processes are independent, the grid system should be able to handle independent jobs, which can simplify the scheduling process.

In this case, where the database does not have to be transmitted frequently through the network, most of the time of the application is spent processing the jobs, thus requiring a mechanism that is able to find the appropriate resources that will process such jobs. Since most devices in the home may disconnect from the grid without warning, the grid system that supports such applications should handle that dynamicity and make the best use of the available resources, which includes providing a fault tolerance mechanism that allows the application to recover from a faulty state without having to rerun all the jobs.

For example, the database described in [20] contains the coordinates of eyes, tip of the nose and centre of the mouth for 130 subjects, in 32 different conditions each (the database also contains a label that identifies each one of these conditions). All the information uses 110 KB of space in the disk, which is a relatively small value to be transmitted via the network. If we consider a similar database representing information about 1000000 criminals, for instance, the amount of memory required for the database would be approximately 800 MB (this number can be reduced using compression techniques – by simply adding the database file to a zip file on Windows, it can be reduced to approximately 260 MB, for example).

The database may be large, but once it has been distributed to the providers, the application will send only a small amount of data. For example, if we assume a scenario where the recognition processes/executables have been distributed with the database, only the image to be analysed (40 KB in the case of the images captured from CCTV cameras used in [20], for example) will need to be sent, plus a set of descriptive information (possibly less than 10 KB).

Since the network usage is reduced once the database is distributed (from 260 MB to approximately 50 KB in the example above), a grid system protocol that does not add much usage of the network itself may help to improve the performance of the application.

This kind of application also requires means to describe the minimal quality of service requirements that should be met by the providers, e.g. the amount of storage needed for the database.

There might be a situation where a new provider with high computational resources connects to the grid and the recognition processes are sent to the grid before this new device has had time to acquire the database. In this case, it might be better that a less powerful provider that already contains the database has priority in processing the jobs instead of the more powerful one. Considering this scenario, for optimization purposes, it may be simpler for the scheduling process if the providers themselves decide if they can process the jobs and are thus themselves candidates to do such processing.

2.1.2 Recommender System

A recommender system for a group of people should be able to retrieve the preferences of the participants in order to provide a good/accurate recommendation. Such information could be stored in a database in the internet, or they could be retrieved from mobile devices carried by the participants (e.g. mobile phones, tablets), which would, most likely, be running on battery power.

This kind of application requires that the grid system not only presents a scheduling and fault tolerance mechanisms, but also that techniques applied do not add unnecessary processing and communication to the mobile devices, since it will cause their batteries to drain more quickly.

Since mobile devices save power by sleeping, there is a definite penalty to polling the client. This way, solutions that avoid client polling are better.

The grid should simplify the registration of the devices in the grid, so that the registration does not require much communication and processing, to save battery on those devices running on it. For example, it is not necessary that the registering device should communicate with all the others in the grid, or find out about all the available services provided by other devices at register action time; it could be done on demand.

In the literature, there are some solutions for computing SVD in parallel, especially for environments with shared memory, like clusters, for example. Some solutions can be found for environments with distributed memory, but it is still an

open research topic to find more efficient ways to solve the problem in such environments.

The computation of SVD in parallel basically consists in dividing the matrix into blocks and distribute the processing of these blocks. Most of the time is spent on running the sub-processes rather than communication – for a r x r matrix block, the cost of data movement is $O(r^2)$, while typically $O(r^3)$ operations are performed using the data [21].

In the case of recommender systems for TV, the matrix can become very large, since there are thousands of media contents that can be chosen, but it may not require much storage or bandwidth to transfer it (normally only the small blocks will need to be transmitted with the jobs). For example, the data set provided by the GroupLens Research Group[3] containing 100000 ratings for 1682 movies by 943 users requires approximately 2MB of storage. The amount of storage may increase with more movies and users, but this database can be pre-distributed similar to the face recognition database presented in the previous section. Once a recommendation is required, only the preferences of the people involved in the decision making process, which can be considerably less than 2MB, especially if one of the user does not have many preferences yet. This poses the requirement for the grid protocol to minimize the network usage not to degrade the performance of the system.

Should a new solution for the recommendation or the SVD problem require the transfer of more data in order to get better results, the network usage by the grid protocols would probably becomes irrelevant.

2.1.3 Health Systems

Some health systems could be implemented following the example of the face recognition systems, where instances of the data to which the biological signs will be matched to are stored *a priori* in the resources that will process the task, so that the communication overhead by the application can be reduced.

Considering the previous statement, we can assume that the requirements of the health system applications are similar to the ones presented for the face recognition system, specially the one related to a fault tolerance mechanism, so that the application does not have to resubmit all the jobs and downgrade its performance.

[3] GroupLens Research: http://www.grouplens.org/node/73

Most of the biological signs (e.g. blood pressure, blood sugar, heartbeat, body temperature, body fat) may be collected and analysed continuously, and the size of the data is expected to be small and not require much bandwidth or storage, concentrating most of the time of the application on the computation of the tasks.

Another scenario for health systems is described in [22], where the biological signs are collected by wearable sensors and sent to a mobile device (e.g. mobile phone, PDA) for storage and display. This allows the biological signals to be collected even when the person is outside the home.

The mobile devices may then act as the consumer of the application, send the jobs to the grid and wait for the results. This brings the requirements for a grid protocol that does not require much communication from the consumers, since mobile devices may be running on battery.

2.2 Power Consumption

Since some of the devices may be running on battery power (e.g. the wearable sensors for the health applications, or mobile phones carrying preferences in the recommender system or submitting jobs to the grid), it is important to minimize processing and communication in such devices, so that their batteries do not become drained quickly. Some of these devices (e.g. sensors) may not be directly participating on the grid, but may use a proxy instead, in which case they may not be affected by any communication overhead generated by the grid protocols. However, other devices (e.g. mobile phones) that participate directly in the grid will be affected by the protocols if those devices are required to send, receive and process many messages.

In terms of power consumption, CPU/memory and wireless interfaces are responsible for a considerable part of it. For example, a Toshiba 410 CDT mobile computer demonstrates that nearly 36% of power consumed is by the display, 21% by the CPU/memory, 18% by the wireless interface, and 18% by the hard drive, as mentioned in [23].

Even though improvements in battery technology have been made, such improvements are not in the same speed/level as the ones for processing power and storage, and it is unlikely that a dramatic solution to the power problem is forthcoming [24].

It is important to understand the power characteristics of mobile radio used in wireless devices in order to design efficient communication protocols.

A typical mobile radio may exist in three modes: transmit, receive, and standby. The transmit mode is the one with maximum power consumption, while the standby mode is the one with the minimum. For example, the Proxim RangeLAN2 2.4 GHz 1.6 Mbps PCMCIA card requires 1.5 W in transmit, 0.75 W in receive, and 0.01 W in standby mode [23]. Power consumption for Lucent's 15 dBm 2.4 GHz 2 Mbps Wavelan PCMCIA card is 1.82W in transmit mode, 1.80 W in receive mode, and 0.18 W in standby mode. These facts should be taken into consideration when designing protocols that involve devices with limited power resources, such as the grid system proposed in this thesis.

In [25], we can find an analysis of the amount of energy used to transfer data using some wireless technologies: Bluetooth, WiFi (802.11) and GSM/EDGE. When considering Bluetooth, it was observed that for a fixed data production rate, increasing the sniff interval causes a proportionate decrease in power consumption; however, for a fixed sniff interval, decreasing data production rate does not cause a considerable decrease in power consumption. As for WiFi, it demands high energy for the wakeup and connection maintenance, while low energy is required per bit transmission and high bandwidth. For the cellular technology (GSM/EDGE), low energy is required for connection maintenance, but high energy is needed per bit transmission and low bandwidth.

In Figure 1 (extracted from [26]), we can see how the power consumption varies with the type of technology and the bit-rate used in the network card for the smartphone Nokia N95. The results show that the higher the bit-rate used, the more energy efficient the data transfer is. We can also see that the 3G technology is more sensitive to the change in the bit-rate ratio.

In the home environment, there could be devices connected using a variety of technologies, and the grid should be generic enough to work with any topology and independent of the technologies in order to simplify its deployment and usage.

Network costs related to power consumption can be classified in two types [23]: communication related and computation related.

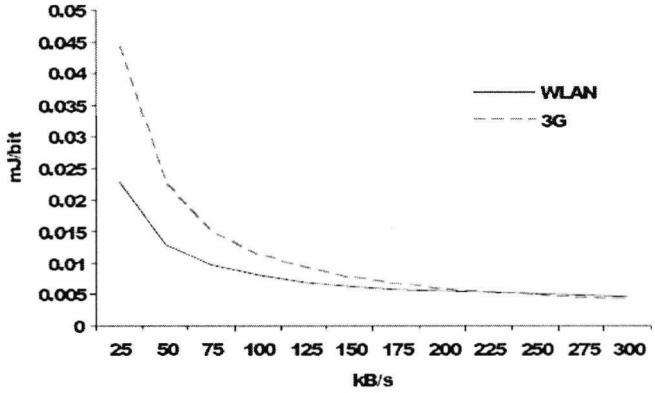

Figure 1 - Energy per bit for Nokia N95 WLAN and 3G.

Communication involves activities in the source (transmitter) and in the destination nodes. In some cases, e.g. ad hoc networks, there is also the cost of being an intermediate. The activities in the transmitter include sending control, route request and response, and data packets (originating at the node itself or routed through it). The receiver is used to receive data and control packets (some of which are destined for the receiving node and some of which are forwarded).

The computation aspects are mainly related with protocol processing, including CPU, main memory, disk and other components usage.

There are some tradeoffs between computation and communication costs. A technique that attempts to lower communication costs may increase the computation ones, and vice-versa. For example, data compression techniques, which reduce packet length (and hence energy usage), may result in increased power consumption due to increased computation. A grid system for the environment envisaged in this thesis should try to find a balance between these two costs.

2.3 The Home Environment

This section presents characteristics of the home environment in terms of devices that can be found in such environment and how they can be set to communicate with each other.

2.3.1 Devices

Designing a Computational Grid for the home environment imposes some difficulties, mainly regarding the discovery and scheduling services, as we can find a

number of different devices that can form part of the grid, and all with different capabilities, especially related to the technologies that allow them to communicate with each other.

In this section, we are going to describe some of these devices, indicating how they can be part of the Grid (as a resource provider or a consumer) and pointing out some possible architectures to group them when building the Grid system.

The first group of devices we can think of when talking about designing such a domestic grid system is the one composed of desktop computers and laptops, which were owned by over 70% (seventy percent) of households in the United Kingdom in 2008, according to [27]. This number is expected to keep rising, at the same time that the number of computers per house is also expected to increase. The other good reason for including these devices in the domestic grid is the fact that they are equipped with good processing, storage and communication capabilities, that can be shared with others devices that lack such good resources.

Another group of devices that can contribute to the grid, by providing relatively good processing power and, eventually, storage capacity, is the new generation of game consoles (e.g. Sony PlayStation, Nintendo Wii, Microsoft X box). They are also supplied with good communication potential to enable multiplayer games through a network (online). The usage of these devices has also increased, and is being incorporated in the home media centres of many families in the United States [28].

With the actual popularity of interactive digital television, we can now find another device that can be part of the proposed grid system: the Set-Top Box (STB). The main duty of a STB is audio and video processing, but they are also capable of running applications. Since it is not a general purpose device, the STB generally has limited processing power and storage, and some may not have any way of communicating with other devices. Nevertheless, these devices have been evolving very fast and it is already possible to find STBs with a large amount of disk space (to store media content) and better processors (not very fast though). The growth of interactive services via the Internet and the development of IPTV (TV transmitted via the Internet Protocol) have forced the introduction of a communication medium in the STBs, usually a V.90 modem or an Ethernet card, which facilitates the inclusion of these devices in the home grid.

Any STB with communication capabilities can be part of the grid, sharing available processing power (especially when not decoding audio and video) and also making use of the resources shared by the other devices in the grid (for example, when executing a recommender system using techniques that are computationally heavy, e.g. SVD). The extra storage that can be found in some STBs can also be employed in the case of developing a data grid.

Similar to STBs, we can also find other limited devices in the home environment that are not able to contribute to the grid with processing power, but can make use of it, and this is the main reason for building the domestic grid, so novel applications can be created for such devices. Mobile phones and Personal Digital Assistants (PDAs) are examples of such devices. However, the latest models of these devices have shown great improvements in terms of processing power and communication (many of them presenting more than one alternative for this: GPRS, 802.11, Bluetooth and Infra-red), which could also be used to process some jobs sent by grid applications. The main drawback of them is the use of batteries, that can be drained very fast when executing such jobs or using certain technologies for exchanging data (using 802.11 consumes more battery than using GPRS, for example).

If we assume that, in the home environment, these devices can also be found plugged into an electricity power source, and they can be very useful to increase the capabilities of the grid system, not just as consumers, especially for the new models to come. Note that not all phones or PDAs in the grid will have the capabilities to run jobs, and they may be running on battery power, a fact that should be taken into consideration for the grid protocol. These devices can be configured to run exclusively as consumers, and discard messages that are not meant for consumers (it is preferable for the devices to do a small amount of processing and identify that the message should be discarded, rather than trying to process the whole message that is not going to lead to any real contribution to the grid).

Sensors are another example of devices that may also be limited by the use of batteries, but in this case, they are also very limited in processing power, storage and communication, which makes them mainly consumers of the grid resources. There may be cases where sensors will not take part in the grid, but only feed data to another device that is participating in the grid.

Because of such limitations, we have the requirement for an efficient discovery and scheduling system that can reduce network throughput required between devices

and the processing required by these activities (discovering and scheduling) and, therefore, take more advantage of the available resources.

There are other devices commonly found in homes that are, at present, generally incapable of processing information, not even to store or transmit/receive it. Examples of such devices are refrigerators, freezers and microwaves. In a near future, it is expected that this kind of equipment will have some extra capabilities and be able to process information and, consequently, take part in the grid, acting as a consumer or even as resource provider.

As some of them are always plugged into an electricity source (and thus most likely to be available in the grid), it would be interesting to provide such devices with more processing power, or even storage capacity, so that we could use them to provide persistent services and so make better use of electricity.

Other devices like printers, scanners, cameras and TV sets, for example, would be helpful for input and output of information. If these devices acquire other kinds of resources, they can also be made available in the grid.

Considering the devices mentioned above, we can see that a grid system for the home environment is already possible, and can be a great incentive for the development of novel applications for the limited devices present in the house. In the near future, we expect new devices to be able to add value to this grid.

2.3.2 Communication Aspects

Here we introduce the different ways of grouping the devices that we are considering, aiming to find those that could produce good results.

The first model to be considered consists of letting one of the devices be responsible for receiving requests from all the other devices, scheduling the jobs and coordinating them. This central point is also required to keep a database with the information about resources and services available in the grid. Figure 2 shows this centralized model.

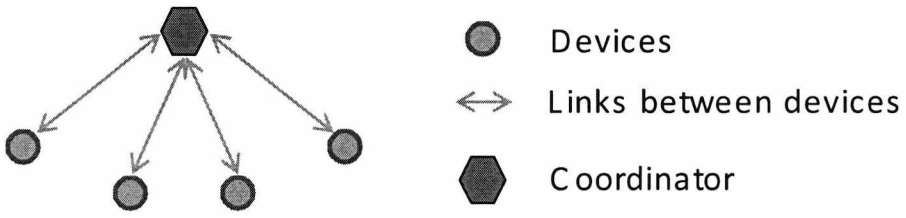

Figure 2 - Centralized Model

This model requires a device to be always on, so the others can interact with it to make use of the grid infrastructure. This model introduces a single point of failure that can compromise the whole system. Nevertheless, this kind of model benefits limited devices, freeing them from additional processing and communication, since the central device does most of the processing.

The grid scheduling system must be efficient in order not to overload the central device in order to provide good performance and to allow this device to contribute to the grid with processing power, if it is not limited and has enough capabilities to do so. It also should be designed in a way that it does not require too much communication from the limited battery-operated devices that send jobs to the grid.

An alternative to the centralized model is the decentralized model, where all devices communicate with each other (see Figure 3), sharing the responsibility for finding resources, scheduling and coordinating jobs.

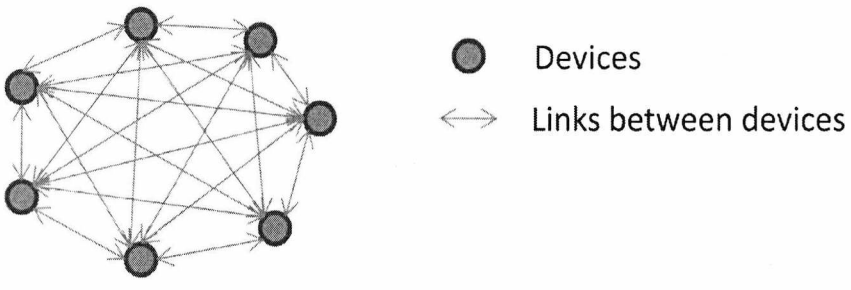

Figure 3 - Decentralized Model

The first advantage of adopting a decentralized model is the exclusion of the single point of failure, found in the centralized model, since it does not have to rely on a single node to coordinate the whole system.

The distribution of coordination tasks between all devices might not be the most appropriate for a home grid for the applications mentioned earlier in this chapter. Depending on the algorithms and techniques used to perform such tasks, this model

would be the best option for applications that would not involve very limited devices.

As an attempt to avoid this unnecessary processing in the limited devices, especially those that act only as resource consumers, devices can be grouped in a hierarchical model, where some of them interact with the core of the grid via another device, with better resources, acting as a gateway. Figure 4 shows a hierarchical centralized model in which the core of the grid is organized as a centralized model and some devices communicate with the grid via devices in the core. Some of the devices may also be organized into a sub-network (e.g. a sensor network) that also communicates to the main grid via a gateway.

A grid that follows a hierarchical centralized model would require an efficient fault tolerance mechanism, because, despite its benefit, this kind of model adds another point for failure: the gateway. Failure in this component may prevent devices from accessing the grid, unless these devices can adapt to use another gateway for this communication.

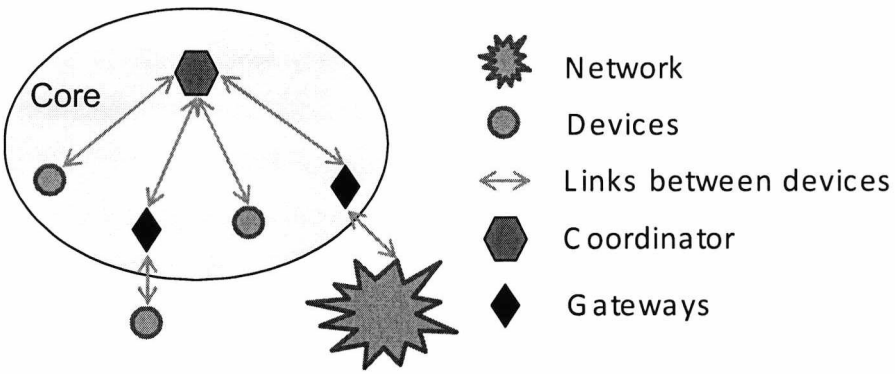

Figure 4 - Hierarchical Centralized Model

The Hierarchical Decentralized Model (see Figure 5) is very similar to the previous model, except that it has the advantage of eliminating one of the points of failure: the central device. It also presents the gateways as possible points of failure, which can be solved with an adaptive mechanism, as mentioned before.

All the models have their own pros and cons, and it is difficult to say which one is the best. It will depend on the chosen information services (discovery, scheduler and coordination), i.e. on the technologies and algorithms they adopt, and

also on the type of applications and devices that are targeted to benefit from the system.

It is difficult to define a fixed topology for the home grid, since it will depend on the devices available in each house. In any case, we need to specify a good fault tolerance model that defines what behaviours the devices in the grid must follow if the system fails or becomes disconnected (whether planned or not).

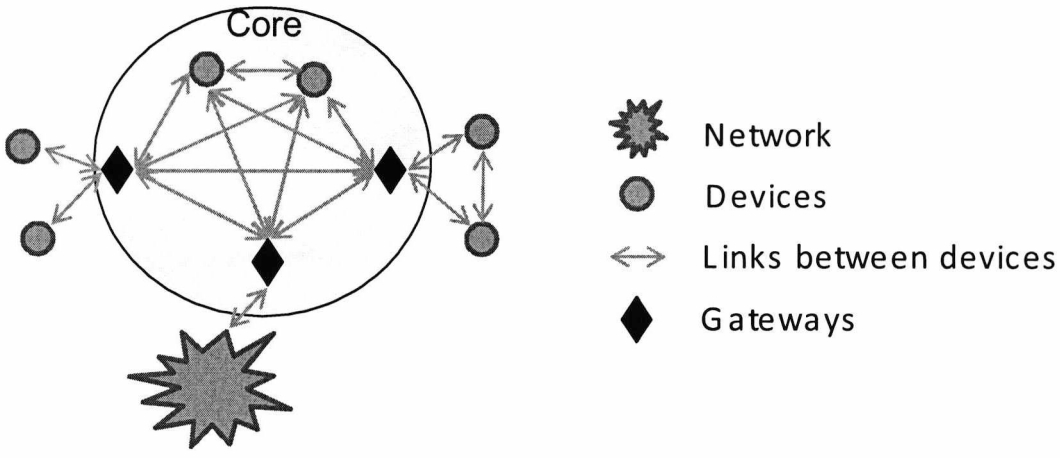

Figure 5 - Hierarchical Decentralized Model

If we cannot find equipment which is connected to the home grid most of the time, other mechanisms that suit the current topology must be defined for the management of the grid. Since the topology of the home network may change very often with devices connecting and disconnecting, the grid must be able to adapt itself to the new topologies, taking into account the limitation of the devices, avoiding unnecessary processing during the adaptation process.

We will return to this categorization when discussing the selection of scheduling mechanisms in Chapter 4.

2.4 Zeroconf and Universal Plug-And-Play (UPnP)

The configuration of network devices seamlessly and without any user intervention is a principle of the Zeroconf[4] and the UPnP[5] techniques. They provide mechanisms for devices to register themselves in the network and get their own IP address.

[4] http://www.zeroconf.org
[5] http://www.upnp.org

The Zeroconf is a protocol that defines an IP network without any manual configuration. The network is built by using three main tasks: The first one manages the numeric network addresses for the connected devices; the second one handles the host names for the devices; and the third one publishes and searches services for the devices. Probably the most known implementation of Zeroconf is Apple's Bonjour, which is proprietary. The Avahi is a free software implementation of the Zeroconf.

The UPnP consists of six layers, which are addressing, discovering, description, controlling, *eventing* and presentation (where only the first three layers are compulsory). The addressing layer is the same as in the Zeroconf and is responsible for assigning an IP address to the device.

UPnP was created to facilitate networking in the home and corporate environments, and it defines protocols for devices to join a network, use and publish services using other protocols such as HTTP, TCP/IP, XML and SOAP (for the service requests).

Many manufacturers of technological devices (e.g. printers, computer, TVs) have been adding support for UPnP in their products, but UPnP is based on a more complex configuration. However, there are some research projects (e.g. [29] and [30]) that propose some alternatives to incorporate devices such as sensors into an UPnP network.

If the home network is composed of devices that support UPnP or Zeroconf, it makes sense to use the existing infrastructure for the registration in the grid system. In this thesis, we do not assume that all devices support these technologies, so we propose our own simple registration mechanism, which can be later adapted to use any existing protocol in the future.

2.5 Requirements

Based on the applications described in this chapter, we have the following summary of the requirements for a home grid scheduling protocol:

A. Communication costs:
    1. Low communication overhead for consumers;
    2. Avoid client polling.
B. Computational costs:
    1. Keep the processing in consumer devices to a minimum.
C. Failure properties:

1. Recover from a faulty state without having to rerun all the jobs (this should take into considerations the communication and computational costs requirements above).

D. Configuration management:

1. Simple registration mechanism;
2. Handle independent jobs;
3. Provide means to describe minimal QoS requirements for the jobs.

In Chapter 3 and Chapter 4, we refer back to these requirements when we present a decision that satisfies them.

2.6 Final Considerations

In this chapter, we have presented some applications that can run in the home environment and make use of a grid system to improve their performance.

Even though the requirements that those applications impose to the grid system are not exactly the same, they have many similarities, and this thesis provides a solution that is generic enough to support those applications and that could be used by future applications as well.

One can argue that it is easier to simply buy a powerful computer and leave it on to execute these applications. That would be the case if we have a critical health application that would need a very reliable set of resources. However, it may be financially more difficult for some homeowner to acquire resources that are more powerful and it would be better to have the grid to make better use of the existing processing power.

Chapter 3. Grid Computing

3 Grid Computing

3.1 Introduction

Considering the under-used computational power available in the home environment (computers, game consoles, PDAs, etc.), it is possible to envisage the creation of a domestic computational grid structure. Grid computing [12, 31, 32] basically consists of sharing resources in a flexible, secure and coordinated way. These resources can be storage and processing power, or even specialized facilities such as a TV screen, for example. Some policies and quality of service parameters must be defined to determinate when and how a resource can be used in the grid.

The following description of grid computing is based on a survey of grid technologies [31].

From the point of view of the user, we can find different types of services that can be provided by a grid:

- Computational services: are secure services that allow the execution of application jobs by distributed computational resources. Grids providing these services are commonly known as Computational Grids, and are the kind of grids that this research concentrates on.
- Data services: provide access and management of distributed datasets.
- Application services: permit the application management and provide access to remote software and libraries transparently for the user.
- Information services: these services are related to the extraction and presentation of data acquired by using grid services (computational, data and/or application);
- Knowledge services: services concerning the acquisition, use, retrieval, publishing and maintainability of knowledge to assist the users in achieving their objectives.

Building a Grid requires the development of a number of services, which include security, information, directory, resource allocation and payment mechanisms [31, 32]. Some services for application development, execution management, resource aggregation and scheduling are also required.

Barker et al. [31] identifies that the main characteristics of a Grid are:

- Multiple administrative domains and autonomy: the resources may be distributed in multiple administrative domains and each resource's owner has the autonomy to manage it.
- Heterogeneity: the resources can be based on a great variety of technologies.
- Scalability: a grid may be composed of a few integrated devices or of millions.
- Dynamicity or adaptability: as the possibility of failures is high in a Grid, because of the large number of resources that can be present, the resource manager or applications must adapt to use the available resources and services efficiently and effectively.

The main components of a Grid are: (i) Grid fabric, representing all the available resources (operating systems, libraries, protocols, computers, networks, etc.); (ii) Core Grid middleware, which refers to the core services of the Grid (security, discovery, storage access, QoS aspects such as resource reservation and trading, etc.); (iii) User-level Grid middleware, including the services to support application development, execution management, resource aggregation and scheduling; and (iv) Grid applications and portals, which consist of the applications that run in the Grid environment.

Computational Grids were designed for applications that are expensive computationally. According to Vraalsen et al. [33], an application is suitable for grid environments if it has a high ratio between computation and communication, which has complexity at least of the order of $O(n^2)$. Since SVD has complexity in the order of $O((m + n)^3)$ [6, 34] (where m and n are the number of rows and columns of the matrix, respectively), it can be seen as a potential application to run in a grid.

Nowadays, we can also see computational grids as a mean to extend the capabilities of limited devices and help them perform complex applications such as the ones mentioned in the Chapter 2.

There are plenty of efforts in developing Grid infrastructures, as presented in [31], but most of them are not suitable for limited devices in terms of processing, bandwidth and storage.

From the initiatives presented in Table 1 (which compiles some of the most well known conventional grid technologies), Globus [35] is possibly the most used technology by the grid community, also described in [36] as the *de facto standard in the Grid computing community*. Globus provides generic services to address issues such as security, resource discovery, resource management and data movement; and these services are often used for the development of new services inside the grid. For example, many grid schedulers (also referred to in the literature as meta-schedulers) make use of the Grid Security Infrastructure (GSI) provided with the Globus Toolkit (GT). Globus itself does not provide a grid scheduler.

Table 1 - Examples of Grid efforts

Initiative	Focus and technologies developed	Category
Globus	Basic software infrastructure for computations that integrate geographically distributed computational and information resources – www.globus.org.	Core Middleware and Toolkit
Legion	Supports transparent scheduling, data management, fault tolerance, site autonomy and a wide range of security options – legion.virginia.edu	Core Middleware and Toolkit
AppLeS	Application-specific approach to schedule individual parallel applications – apples.ucsd.edu	Grid Scheduler
Harness	Builds on the concept of virtual machine and explores dynamic capabilities. Focuses on parallel plug-ins, P2P distributed control and multiple virtual machines – www.epm.ornl.gov/harness	Programming environment and runtime system
WebFlow	An extension of the Web model that can act as a framework for wide-area distributed computing – www.npac.syr.edu/users/haupt/WebFlow/demo.html	Application runtime system
GrADS	An adaptive programming and routine environment – hipersoft.cs.rice.edu/grads	User-level middleware
JXTA	Provides core infrastructure that is essential for creating P2P computing services and applications – www.jxta.org	Core middleware

Another Globus service used by grid schedulers is the Grid Resource Allocation and Management (GRAM), responsible for coordinating the jobs (submitting, monitoring and controlling). If an application does not need state management, executing only input and output data, GRAM may not be appropriate, since it may

delay the response of the application. In this case, it is recommended that the possibility of developing an application-specific service should be considered.

It is important to point out that GRAM is not a resource scheduler, but an engine that can communicate with local resource schedulers via a standard message format. GT already provides interface implementation to some resource schedulers like the Condor and Portable Batch System (PBS) schedulers, for example. By default, GRAM uses a "fork scheduler", i.e. it just sends a new process to be held by the operating system.

Similar to Globus, Legion [37-39] also provides an infrastructure based on objects for resource sharing in a grid. It has an object called the Enactor, which is responsible for the execution of the tasks once they are scheduled. The Enactor also allows resource reservation. Legion also supports the addition of new scheduler objects, but, in contrast to Globus, Legion provides a default scheduler, which is very simple and schedules jobs randomly without considering any QoS issue such as load, CPU power or bandwidth. Chapin et al. [38] presents an improved version of the default random scheduler for independent tasks. Both scheduling solutions involve querying a Collection object for available resources. The Collection object contains the information about all the resources registered in the grid.

According to the "Lessons Learned" section in [39], the flexibility added with the separation of the scheduler and the Enactor objects was never used, because complex scheduling techniques that require reservation are useful only for a small set of applications.

An important grid initiative that is not present in Table 1 is OSGA [40], a service-oriented architecture that standardizes core functionalities and behaviours of grid system. The standards include security aspects such as identification, authentication and access control policies; it also covers the discovery of services, description of jobs and the management of their lifecycle, including exception handling.

The OSGA standards are based on web services specifications (e.g. WSRF, WSDL, UDDI, etc.). In fact, the newest versions of GT are implementations of OSGA.

These technologies require a big infrastructure of powerful devices to run the grid without suffering with performance issues. In our case study of a home grid, we assume that such an infrastructure does not exist and it is difficult to guarantee

resources to run components such as the Collection, in Legion, or the GRAM or GSI in Globus. Our distributed scheduling solution does not require such an infrastructure and does not require devices to know about each other, and still provides good performance and fault tolerance.

The next section presents a brief overview on computational grid solutions for limited devices.

3.2 Computational Grid for Limited Devices

Some work has also been developed in terms of grids for limited devices, as in [10, 11, 13, 41-44]. Most of these papers focus on allowing limited devices to use existing powerful grid infrastructures, which has a different scope from this thesis. The closest work to that presented here is [13], where a grid with home devices (called embedded-Grid or e-Grid) is presented. The main objective of the grid in [13] is to export the power of embedded devices in the home to an external grid, with a powerful device (usually a PC) acting as a gateway that communicates with the external grid server and manages the embedded devices.

In the e-Grid, the gateway is also responsible for deciding if a certain embedded device can participate on the grid and receive jobs to be processed. This is done by checking if the overall performance gain from all devices together is greater than the overhead of monitoring activities and communication latencies. For this purpose, the gateway computes a degradation factor for the addition of a new device, and based on this factor it is decided whether the new device can be part of the grid or not.

Generally, the external grid server generates all the workload for the e-Grid. In this thesis, we are mainly interested in a workload generated by the limited devices in the house, and we expect a device with low capabilities to act as the central device, instead of a powerful device as in e-Grid.

Ahuja et al. [10] presents a survey of grid technologies for devices in wireless networks (Wireless Grid), and groups them in three categories: (i) Sensor Networks and Grids, (ii) Fixed Wireless Grids and (iii) Mobile Wireless Grids. Research has been done in terms of grids for sensor networks and mobile wireless grids, which can bring ubiquity to the grid.

In this thesis, we are interested in developing a computational grid in the home environment, which can have characteristics from the three categories of wireless grid mentioned above: a house may have sensors to identify presence of someone or

to check temperature or for smoke, for example [category (i)]; people have been adopting fixed wireless networks at home for connecting to the Internet though a wireless router, and the connected devices inside the house are expected to be part of a local network [category (ii)]; a person may want to use his/her grid at home from the mobile phone, for example, while he/she is away from home [category (iii)]. In this domestic grid, we can also find devices connected via a wired network.

3.3 Scheduling

Scheduling is a very important factor in computational grids in order to allow them to provide good performance results. In particular, scheduling can be the mechanism for maximizing the resources available to end users and to exploit idle resources [45].

Although several efforts have concentrated on developing scheduling systems, most of these systems schedule jobs within a single machine or across clusters of homogeneous machines.

In the context of grids, single machines and clusters represent resources that can be used to process jobs, and they possess their own local scheduler. Thus, we can identify the need for higher-level schedulers that can have a general view of the resources and are able to communicate with their local schedulers in order to send jobs to them. These higher-level schedulers are commonly referred in the literature as meta-schedulers and they are responsible for handling the heterogeneity of grid environments. In this thesis, the terms scheduler and meta-scheduler are used interchangeably, referring to the same component.

Figure 6 (extracted from [45]) shows an example of the interaction between a meta-scheduler (usually a centralized component) with local schedulers in conventional grids running through the Internet.

A meta-scheduler is usually responsible for managing the access to resources and the load balance between them.

According to Adzigogov et al. [45], meta-scheduling is composed of three stages: Resource Discovery, System Selection and Job Execution. Each stage contains sub-stages, but meta-schedulers are not obliged to have all of them.

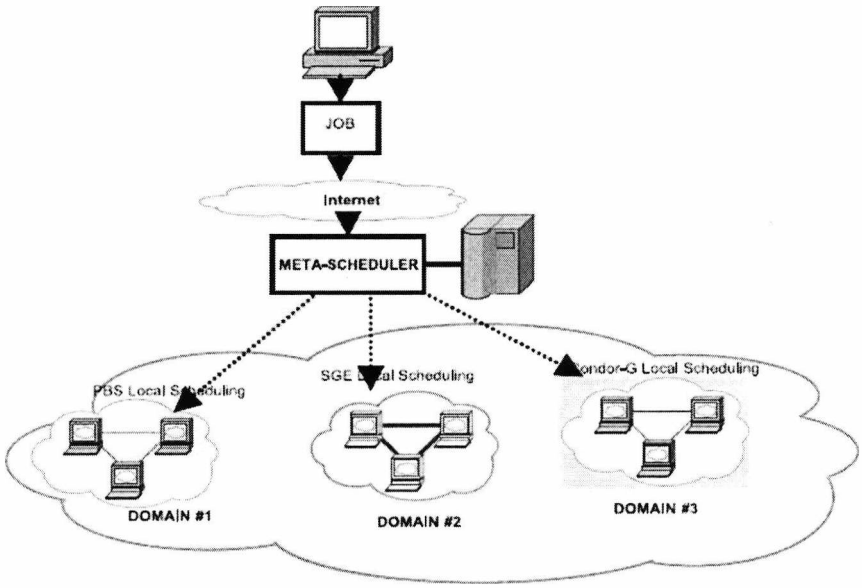

Figure 6 - Interaction between meta-scheduler and local schedulers.

3.3.1 Resource Discovery

It is during the resource discovery phase that the scheduler defines a list of resources that can run a certain job. Resource discovery includes three sub-phases:

- *Authorization Filtering*: at this stage, the resources are selected based on the privileges of the job owner. If the owner has no authorization to run a job using a certain resource, that resource is not considered to run the job.
- *Application Definition*: this is where the requirements for the job are specified in terms of minimum CPU power, storage, memory, etc. Some smart compilers or other tools have been used [45] in order to provide a more accurate and complete definition of these requirements.
- *Minimum Requirements Definition*: this sub-stage uses the requirements defined in the previous sub-stage in order to find the most capable resources for running the job. It is usually combined with the steps of the system selection stage.

3.3.2 System Selection

Once the set of candidate resources is defined, one of them must be selected to run the job. This selection happens during this stage and is performed in two steps:

- *Information Gathering*: here is where the meta-scheduler gathers information about the resources in order to find those that match with the job

requirements. This is usually done by sending queries to a centralized service that maintain information about all the resources in the grid, like the Monitor and Discovery Service (MDS) from the Globus Toolkit, for example.

- *System Selection*: this stage is where the final decision about which resource is going to run the job is made. This decision can be random, or based on the information gathered (e.g. host's load and memory), on estimated information (e.g., job execution time) or on some optimization criteria (e.g., minimizing the execution time or fragmentation of resources) [45]. Once the selection is complete, the meta-scheduler starts preparing the job to be submitted to the chosen resource. Section 2.3.4 presents a discussion about the algorithms for the selection of the resources.

3.3.3 Job Execution

Once the matchmaking process is complete, it is time to run the job. This can be done using the following steps:

- *Advance Reservation*: reservation of a resource is only possible if supported by the local scheduler, since it is the only one with control over the resource. Reservation can be very useful in cases where a required resource is most likely to be very busy all the time, or when it is required for future usage like a demonstration, for example.
- *Job Submission*: this is the stage where the job is sent to the selected resource's local scheduler for execution.
- *Preparation Tasks*: in this step, the binary executable job and the required input and output data are moved to the designated places.
- *Monitoring Progress*: this is the stage where the running time of the jobs is verified; depending on the application, if the job is taking too long to run, it may be rescheduled, which can be significantly costly on grid systems.
- *Job Completion*: at this step, the job's owner (consumer) is notified of the job completion. Moreover, this step can be very difficult to perform because of the high probability of failures in the grid. Ensuring job completion notification is an open research topic.
- *Cleanup Tasks*: this step consists of the removal of any temporary local settings that have been defined for the job execution. It also includes any

communication with the resource in order to gather information for data analysis regarding the execution of the job. The cleanup process is also executed on the hosts for the binary, input and output data.

In the next section, we describe some scheduling algorithms, applied in both grids and other distributed systems.

3.3.4 Scheduling Algorithms

Dong and Akl [36] and Krauter et al. [46] present taxonomies and surveys on scheduling; they also present open issues regarding grid scheduling. Krauter [46] focuses on classifying grid scheduling systems according to the taxonomy it proposes, without mentioning the algorithms used internally by those systems. Dong and Akl [36] focuses on the scheduling algorithms and approaches used by distributed system, specially computational grids.

Figure 7 shows the taxonomy of grid schedulers based on the dependency of the tasks proposed by Dong and Akl [36].

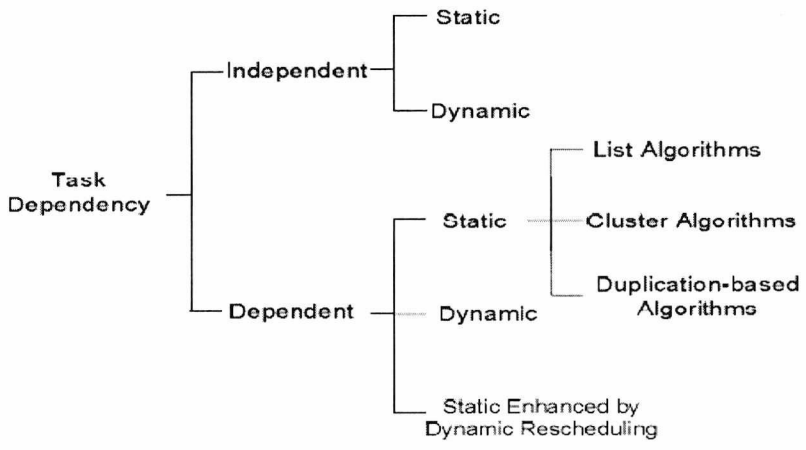

Figure 7 - Task dependency taxonomy of Grid scheduling algorithms.

According with this taxonomy, we have two main categories of schedulers when considering the task dependency: those that deal with independent tasks and those with dependent ones (according to the requirement D.2 in section 2.5, home grids need to be able to process independent jobs).

Improvements to the grid infrastructure have allowed better support to dependent tasks by the schedulers [36], and they usually use directed acyclic graphs (DAGs) to

model the jobs, where the vertices represent the jobs and the edges represent the precedence between jobs. Most of the systems in this category use static mechanisms, which does not deal well with the dynamic change of the resources in the grid (which does not comply with the requirement C.1 in section 2.5).

For grid schedulers that attempt to optimize the performance of a particular application (e.g. AppLeS [47]), many static heuristic algorithms can be used (e.g. Min-Min, Max-Min, XSuffrage, MET, MCT – more detail about these algorithms can be found in [36, 48, 49]). Such algorithms usually use predicted performance of the resources in order to allocate the tasks. In highly heterogeneous and dynamic environments such as the home environment, the performance of these algorithms is less effective, since the static prediction may not reflect the grid current state when the tasks are sent for processing.

Silva et al. [50] presents a static solution for scheduling independent jobs without performance predictions and based on replication, where tasks are scheduled to more than one resource in order to improve the performance. This solution works fine on environments where resources are abundant, which is not the case of home environments.

Dynamic schedulers are suitable for scenarios where it is difficult to estimate the cost of applications or when jobs arrive dynamically at the scheduler. Condor [51] and Legion are examples of systems that use dynamic scheduling, but they both require an infrastructure that cannot be guaranteed in the home environment, in order to get information about the state of the resources.

Dynamic schedulers often make use of load balancing techniques to improve the performance. The four basic approaches to achieve dynamic load balance are [36]:

- Unconstrained FIFO (First-In-First-Out): also known as Optimistic Load Balance (OLB) [48], this approach attempts to keep the balance by assigning the jobs to the next available resource. If more than one resource is available at the same time, one of them is chosen arbitrarily. OLB is one of the easiest grid schedulers [52], but it does not provide optimal results;
- Balance-constrained: this approach assumes that resources receive more than one task, and attempts to rebalance the loads on all resources by periodically shifting waiting tasks from one waiting queue to another. It

can be very costly in terms of communication, requiring adaptive rebalancing heuristics to improve scalability and performance;

- Cost-constrained: this approach is an improved balance-constrained approach. It considers the communication costs in order to decide if a job should be moved from one resource to another;
- Hybrid: this approach mixes static and dynamic scheduling, by applying static scheduling to scenarios where the tasks are certain to be executed, and dynamic to the others.

3.4 Fault Tolerance

In this section, we give an introduction about fault tolerance and describe how it is addressed in grid systems. Fault tolerance is one of the requirements identified in section 2.5 (C.1).

3.4.1 Introduction to Fault Tolerance

Stand-alone systems are prone to failures, and so are distributed systems like computational grids, for example. The difference is that in the case of distributed systems, a failure does not mean that the entire system suffers a breakdown, and it can continue working and recovery actions can take place on the affected parts of the system.

Fault tolerance, in the context of this thesis, is the capability of the system to run continuously and reliably execute jobs in spite of failures [53], and plays an important role in the success of distributed systems, including computational grids.

According to Jin et al. [54], failures in distributed computing systems can be divided into three categories:

a) Node crash: participants in the grid (nodes) may shutdown or stop working correctly for a number of reasons (e.g. hardware damage, battery drained, switched off by the user, etc.);
b) Network failure: a link between two nodes may stop working or the network can be overloaded, for example;
c) Process fault: tasks being processed may fail because of a bug in the code, or because it is allocated to run on a provider with inappropriate resources for the execution of the job (e.g. insufficient storage,

incompatible operating system). Cases where tasks take too long to be completed because the provider's CPU is highly loaded also fall in this category.

Although there are various techniques that can be used for detecting and correcting faults in distributed systems [54, 55], many of the current grids have one or more components that are not fault-tolerant [56].

3.4.2 Techniques for building fault tolerant systems

The main techniques for building fault tolerant systems are:

- Timeouts [54, 57]: network timeouts can be used to detect node crashes and network failures (although they cannot distinguish the type of crash); consumers and brokers can determine timeouts for task completion in order to find process faults (but it cannot determine if a bug happened or if the provider is overloaded, for example);
- Unreliable Fault Detection (UFD) [54, 55, 57, 58]: the most general of the available techniques, this consists of having all components sending "*I am alive*" messages to each other. If after a certain period a component C1 does not receive a message from C2, C1 then adds C2 to its "suspicious" queue, which stores all the components that C1 "thinks" may have crashed. Since it is difficult to determine if C2 really crashed or, if it is just too slow, the method allows C1 to make mistakes about the status of C2, thus the name of the mechanism. When C1 receives the message from C2, C1 updates its own "suspicious" queue by removing C2 from it and increases the time to wait for the messages from C2. The UFD algorithm presented in [55] assumes the use of Reliable Broadcast, where all broadcast messages sent by correct processes are guaranteed to be delivered, which increases the overhead on the communication; this is different from our proposed mechanism, which can be implemented with UDP broadcast;
- Replication [56-58]: this technique consists of making replicas of a service (e.g. scheduler and replication server) or running a process in more than one server, so that if one of them crashes, the other can be used

instead, keeping the systems working normally. An example of the use of replication is a grid system where a consumer/broker sends two different versions of a task (one that is more efficient with some resources, and another that works better on others, for example) to the grid, and stops the execution of one of them after receiving the result from the other. Data replication can also be used in grid systems, so that the resource selected to execute a task can get the required data from the nearest position.

- Transactions [56, 57, 59]: considering a group of activities to be executed, the system has the ability to rollback to the state before the start of the execution of the group, in the case of some failure of one of the activities;
- Retrial [57, 58]: this technique consists of restarting the process from scratch once a crash is identified;
- Checkpoints [57-60]: during the execution of a task, checkpoints are sent periodically to another server, so the task execution can be resumed if the previous provider executing it crashes. This technique is very useful when there are long tasks being sent to the grid, where restarting it from scratch in case of failure could degrade the performance. Some systems may assume that the checkpoint server will never fail, while other may apply replication of this server.

The choice of what technique to use depends on many factors such as the objectives of the system, implementation issues, target applications and computing infrastructure [58]. For this reason, different grid systems adopt different solutions. The following sections present the support for fault tolerance that some systems provide, starting with Globus GRAM, which is extensively used by other systems.

3.4.3 Fault Tolerance in Grid Systems

Hwang and Kesselman [57] describe the Globus GRAM protocol as being too generic, since it was designed to be used by any meta-scheduler system, which would make the support of fault tolerance more difficult. Although GRAM provides mechanisms to monitor the status of the nodes, and to attest whether the node is still active or not, GRAM cannot detect if a certain job has been completed or failed. For

this reason, grid applications that rely on GRAM for tasks execution (e.g. Condor-G, CoG Kit, Nimrod-G, Ninf-G), often ignore fault-tolerance, or end up creating their own mechanisms, which usually cannot be reused.

Because of this lack of a fault recovery system in Globus GRAM, Hwang and Kesselman [57] present a general framework for fault tolerance. It consists of a failure detection service (FDS) and a failure handling framework (called Grid-WFS). With the FDS, it is possible to identify both application crashes and user-defined exceptions by using a specific notification mechanism. Usually, grid systems do not provide any support for user-defined exceptions. With Grid-WFS, the user can specify a policy for how to recover the operation from a crash (e.g. "retries at most 3 times" or "replicate the operation at hosts X and Y"). Grid-WFS uses the concept of workflows to achieve its generic failure handling mechanism and to separate it from the application code.

Most grid systems use timeouts or UFD in order to detect failures, so they can usually only identify host crashes or network failures. However, distributed systems like NetSolve [61], DOME [62] and PVM [63] use some system-specific polling, notification or generic heartbeat mechanisms, similarly to FDS, to allow the identification of a failure in the application.

Once a failure is detected, fault-tolerant systems must take some action in order to deal with the crash and keep the system working. Condor [51] provides fault-tolerance by using checkpoints, where applications can use the Condor Checkpoint Library. The Condor system assumes the existence of a reliable *checkpoint server* to store all checkpoints. The main limitation of Condor is that its checkpoint library is restricted to some platforms. In contrast to Condor, Condor-G [51, 64] uses retrial on the same machine.

The MAG system [58] providers a portable checkpoint mechanism by using Java technology. MAG also supports the use of replication as an alternative to provide fault tolerance.

The fault handling mechanism adopted by OurGrid [65] system is replication. OurGrid also offers an infrastructure to manage checkpoints by using third party components (e.g. the Condor checkpoint library).

NetSolve uses retrial in the first available machine to handle faults [66].

GridTS [56, 59] relies on a tuple space (TS) to work, therefore it specifies the replication of the TS in different machines, so if the current TS fails, its replica is

used instead. To avoid an inconsistent state of the TS in the case where a broker or resource crashes, GridTS uses transactions. Finally, GridTS supports the use of checkpoints to allow tasks to resume their execution in other devices in case of failure of the original resource processing them.

Another grid framework that uses TS is presented in [67], but differs from GridTS, in that no fault handling mechanism is specified.

Because, in conventional grids, tasks usually take a long time to be executed, checkpointing appears to be a good choice to improve the performance of the system, and has been supported by most of the fault-tolerant grid systems. The use of checkpoints and replications assumes the existence of a reliable infrastructure in order to keep the good performance that these techniques can provide. This kind of infrastructure can easily be found in conventional grids, but is not the case in the home environment that we are assuming for our grid system. In the next chapter, we introduce the fault detection and handling mechanisms adopted for our system.

3.4.4 Network Partition

Network partition is a phenomenon where a computer network splits, usually as a result of an error, into more than one sub-network that continues to work independently [68, 69]. Figure 8 illustrates the partition of a network A into sub-networks A' and A''.

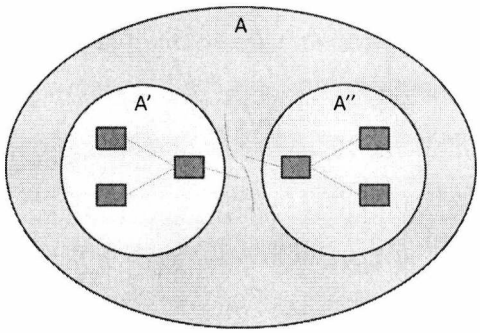

Figure 8 - Example of a network partition.

After the partition, devices in A' can only communicate with the others in A', losing their communication with those devices in A''. This kind of situation may generate inconsistencies between servers that are now in different sub-networks.

When replicated services such as the scheduler or checkpoint server, for example, end up in separate sub-networks, the grid system should guarantee the

consistency of the services/servers when the partitions merge again, which can be a very difficult task.

To avoid this problem, grid systems usually assume that the core services/servers are located inside a fault-tolerant network topology that provides reliable broadcast and devices are connected to the network via at least two interfaces, which could avoid the partition of the network. Jin et al. [54] specifies that a fault-tolerant network topology is a requirement for robust grid systems. Perhaps this is the reason why the partition of networks is not usually addressed in research papers about grids.

Out of all the research papers about grids that have been studied for this thesis, only [70] mentions network partition as a potential problem, and it uses a majority consensus protocol in order to ensure that only one core of the system exists when a partition happens.

Home grids are expected to function with fewer resources than conventional grids, and they are also expected to present higher network heterogeneity, including devices that can be connected through ad-hoc networks, which are highly prone to suffer network partition due to the dynamic change in the topology [71].

Our proposed system does not assume the existence of a fault-tolerant network topology and it can be implemented using UDP broadcast, which does not guarantee the delivery of the broadcast messages. In the case of a network partition, it is possible to have more than one instance of our system running, one in each sub-network, since the participants may elect a new scheduler when it is detected that there is no scheduler in their sub-network.

In Chapter 3, we present a description of how our system identifies that the partitions have been merged, and what the procedures to maintain the consistence of providers and scheduler are.

We have used model checking to verify that our system continues to work when the network is partitioned and after it is merged. The model is presented in Chapter 6, which explores two scenarios of network partition.

3.5 Summary

In this chapter, we made a brief introduction to computational grid technologies (for both powerful and limited devices), and we then concentrated on the scheduling and fault handling in grids, which are the focus of this thesis.

We also exposed the pros and cons of the most common fault tolerance mechanisms to constrained environments such as the one assumed in this thesis; and we described how some grid systems use such mechanisms.

At the end of the chapter, we made a discussion about the viability of building a home grid and the pros and cons of network topologies for the communication between the devices in the house.

Chapter 4. Proposed Home Grid

4 Proposed Home Grid

Peer-to-peer (P2P) systems have to deal with participants that remain disconnected for long periods, which is different from what happens in conventional grid systems [72]. However, that work envisages that large-scale grid systems will develop to present this characteristic of current P2P systems, meeting a demand for the grid to adapt to the higher dynamicity of resources.

This high dynamicity is already a characteristic that can be found in the domestic grid being defined in this work, since most resources may be kept switched off for most of the time. Domestic grids are not only dynamic, but can also be very heterogeneous, being composed of devices with different capabilities and also different network technologies. These devices can vary from a refrigerator with low processing capability connected to the grid via Ethernet to a mobile phone or a computer connected via wireless technologies, for example. The devices in the house that would most likely have the "always on" characteristic (switched on for most of the time) would be a refrigerator, a microwave or a router, for example.

Gradwell [73] investigates three of the most mature scheduling systems for grid computing: Nimrod-G[74], GrADS [75] and Condor-G [51]; and presents a description of similarities between these systems which are dependent on their centralized control. These similarities are:

- The assumption of one central point in the grid, which allows them to know about all the jobs on all nodes on the grid.
- The centralized control (full or partial) of the scheduling policies for all the nodes.
- The assumption that the resource allocation for the jobs and the job scheduling are done by the same part of the scheduler.

If it was possible to guarantee that there is a device, powerful enough to run a central scheduler without compromising the performance of the grid, and that this device is switched on all the time (or most of time), a centralized system like the ones mentioned above could be used for the domestic grid. The devices with the "always on" characteristic will probably not have the required processing power to

support sophisticated fully centralized scheduling systems in the near future, so adopting this kind of system in a domestic grid would degrade performance, since the centralized scheduler demands time for distributing the tasks, and implementing it on low processing power devices would delay the distribution of the tasks.

In conventional grids, centralized schedulers gather information about the load status of the resource to try to promote good load balance for the grid system, adding more traffic to the network. Furthermore, when a scheduling system is described as distributed, it means that a central scheduler for a particular group of resources can communicate with other central schedulers and they can send tasks to each other, without really distributing the scheduling activity.

Research like that presented in [76] and [77], for example, uses the concept of multi-agent systems to distribute the work of the scheduling system. In [76], scheduling agents communicate between themselves using coordination messages to find a global scheduling solution after each one of them have found the best local solution. In [77], agents are used for resource discovery purposes, where each agent has a cache with information about some other agents called its "neighbours"; when an agent does not have information about a particular resource, it seeks the information from its neighbours, which involves a big communication overhead. In this thesis, we also use agents for the resource discovery and scheduling mechanisms.

Inspired by multi-agents solution, in this thesis, the idea is to create simple components that do not require much processing power from the devices where they are running, and to reduce the communication overhead. The communication between these components is done mostly by broadcasting small messages and communicating with a central device that does little processing to make the final scheduling decision.

The remainder of this chapter describes our proposed solution for grid systems running on home environments.

We start with the description of how the system should behave without considering failures. Only in section 4.5, we describe the additional behaviour for the components in case of failures.

4.1 Scheduling Architectures

In Chapter 2, we have described some applications that could make use of a home grid, and found that most of the applications would use the grid to improve their performances, which would make a reliable grid system the ideal world. Since grids are unreliable by their nature (since devices can become unavailable at any time without a warning), a fault tolerance mechanism would be the main requirement for the grid system.

Considering the type of applications mentioned in Chapter 2 and the limitations of the devices in the home network, we have to analyse what is the best scheduling architecture that should be used in our solution. The main possibilities are analyzed in the following sub-sections.

Here we will not discuss solutions where a Network Weather Service (NWS) exists, as in the conventional grid systems. A NWS corresponds to an external service that periodically monitors the status of the devices in the network, instead of the scheduler itself. The ideal case would be to have the NWS running in a different device that is not the scheduler to improve the fault tolerance of the system. This additional device would also be required to be on most of the time, making the NWS not a very attractive solution for home environments, and for that reason it was not considered in this discussion. Another disadvantage of NWS is that they also add more traffic to the network.

For this analysis, we grouped the main solutions found in the literature into three architectures/models: Fully Decentralized, Fully Centralized and Hybrid. There are many ways that grid scheduling protocols can be created using these models. In the following sections, we focus on those solutions that would require the least amount of energy from consumers, considering that they can be running on battery from the requirements in the Chapter 2, and taking into consideration the faults that can happen and the limitations of the environment.

4.1.1 Fully Decentralized Architecture

Considering the difficulty of finding a device powerful enough to run the scheduler without damaging the performance of the system and that is always on, the obvious choice of architecture would be the fully decentralized one, where no central device is required and all the devices communicate with each other.

Figure 9 illustrates the number of messages exchanged between consumers and providers using this kind of architecture. As will see later in this chapter, this architecture requires fewer messages to schedule a job when compared to the other architectures discussed later. In Figure 9, we represent a grid with one consumer and two providers.

- In the step 1, the consumer broadcasts the job to all the providers;
- In the steps 2 and 3, each provider performs the matchmaking and reply to the consumer informing it that they can execute the task;
- In the step 4, the consumer decides which provider will execute the task and sends it to the chosen one;
- In the step 5, the chosen provider completes the job and sends the result to the consumer.

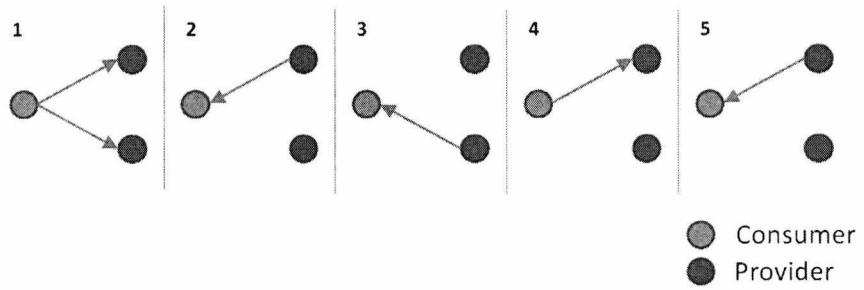

Figure 9 - The fully decentralized scheduling architecture.

The decision making process performed by the consumer can be as simple as picking the first provider who replies (which would save battery for those devices that use it), or to make additional processing to choose the best one based on the characteristic of the providers (e.g. distance from the consumer).

An alternative for the information flow could be the consumers simply announcing that they have a job in step 1, receive the state of the providers in the steps 2 and 3, perform the matchmaking process based on the information received and send the job to the chosen provider. This scenario adds more processing to the consumers, which is not ideal for those running on battery. However, it could be a good alternative for environments where the objective is to optimize the use of the grid for one application and where providers can be reserved.

Even though this architecture uses fewer messages to schedule a job, the amount of power required from consumers will vary according to the number of providers in the grid.

For example, suppose that the consumer sends the job description in Figure 10.

```
message=1
job_id=10
CPU=1500 MHz
Storage=50 MB
OS=Windows
RAM=2024 MB
Power=100 %
```

Figure 10 - Example of a job description.

The size of this description is approximately 696 bits. Assuming that the consumer is a smart phone Nokia N95 and that it spends 0.005 mJ per bit to use the network (as mentioned in the Chapter 2), the power spent to send this job description (Cj) would be 3.48 mJ (the typical capacity for the battery of a N95 is 12600 J).

Now assume that the response from the providers contains only the message type and the job identification to which it refers. The size of the message would be approximately 160 bits (equivalent to a SMS with 20 characters), requiring 0.8 mJ of power from the smart phone to receive it (Cr), for each message received from a provider (0.8 x N mJ of total power, where N is the number of messages received).

Assuming that once the provider is selected the consumer sends the job to the provider, and considering a job of 32768 bits (the size of the source code used for the Linpack benchmark presented in the section 5.3), the cost of transmitting the job (Ct) from the N95 would be approximately 163.84 mJ.

This way, the power required to submit a job (Pj) could be computed using the following formula:

$$Pjd = Cj + (Cr \times N) + Ct.$$

The Table 2 shows the power required to submit a job for different numbers of providers in the grid (considering only the network usage), assuming that all of them reply to the consumer.

Table 2 - Power required from the N95 to submit a job in a fully decentralized model.

N	Power (mJ)
1	168.12
5	171.32
10	175.32

Since the chosen provider may leave the grid without any warning to the consumer while processing a job, we have to consider the cost of handling this kind of fault.

If we assume that the system is not fault tolerant and that a job is resubmitted after the identification of a failure, the energy required just to send the job to another provider would be Ct (disregarding the power required to select another provider). In a scenario where M failures happen before the job is completed, the consumer would have user M × Ct J of energy to retransmit the job to the selected providers. Assuming five failures (M = 5), for example, the Nokia N95 would spend approximately 819.20 mJ of energy only to send the job to the providers chosen to execute the job, which is much higher than the 0 mJ from a fully centralized architecture (see section 4.1.2), or the 17.4 mJ from our hybrid solution (see section 4.1.3).

Broadcasting the job with its description in the step 1 of Figure 9 would consume more network resources, but it would avoid the consumer having to resubmit it to some providers in case of failures (the consumer would still need to send the job to a provider that connected to the grid after the job has been announced).

A common and simple technique for fault tolerance that can be used is the replication of the job, but even in this case the consumer is still required to spend energy sending the job to at least two providers. Furthermore, we assume that it may be difficult to find many resources available in the home at the same time, and this would be a waste of resources.

Note that, in this section, we have only considered the energy costs for the network usage; complex fault tolerant mechanisms may require more processing from the device as well, which also increases the amount of energy used.

Since some applications mentioned in Chapter 2 require fault tolerance and some of them have battery-powered devices as consumers, we decided not to use a fully centralized model for our solution.

4.1.2 Fully Centralized Architecture

In this architecture, the scheduler is responsible for gathering the status information from the providers, perform the matchmaking based on the collect information and choose the provider that should execute a task. This is the architecture followed by most grid scheduling solutions in the literature (including the OLB scheduling heuristic).

Figure 11 illustrates a fully centralized architecture considering one consumer, one scheduler and two providers.

- In the step 1, the consumer sends the job to the scheduler;
- In the step 2 the scheduler sends a confirmation to the consumer;
- The steps 3 to 6 represent the communication between the scheduler and the providers in order to acquire the status information for the matchmaking process;
- In the step 7, the scheduler decides which provider should execute the job and sends the job to the chosen provider;
- In the step 8, the provider completes the job and sends the result to the consumer;
- In the step 9, the provider informs the scheduler of the job completion.

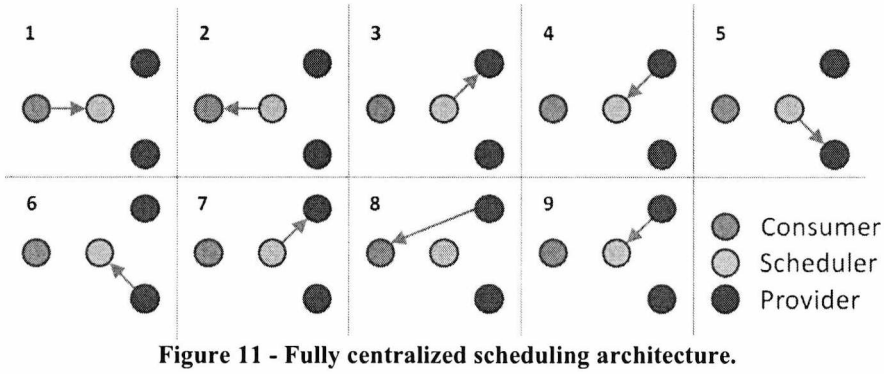

Figure 11 - Fully centralized scheduling architecture.

This architecture uses more network resources than the fully decentralized architecture (4 more messages), but it requires less energy from consumers,

especially for handling faults of providers, since the scheduler deals with it. However, this architecture makes it more difficult to recover from failures of the scheduler, requiring the consumer to resubmit all the jobs again. Furthermore, this solution requires the scheduler to know where the providers are and keep checking if they are still alive.

Following the same reasoning used in the previous section, we can compute the amount of energy that the consumer requires to send a job (Pjc) by the formula:

$$Pjc = Ct + Cr.$$

This formula corresponds to the cost of sending the job to the scheduler (step 1 of Figure 11) plus the cost of receiving its response.

Considering the Nokia N95 as a consumer, the total amount of energy for the consumer to submit a job would be 164.64 mJ, independent of the number of providers.

In the case where the scheduler deal with timeouts for the job completion, the consumer may not be involved in the fault tolerant procedure, so it would not spend additional energy. However, some energy will be spent if the consumer is involved in that process (e.g. managing timeouts and sending reminders).

Some centralized systems usually assume that the central device is fault tolerant, usually by using replication of the scheduling service. In the home environment that we envisage for this thesis, it may not be possible to guarantee enough resources to use the conventional techniques to make the scheduler fault tolerant.

Even though the scheduler may fail, we expect that this type of failure happens less frequently than the provider's failures, since we can adopt election mechanisms that can chose the most reliable devices to act as the scheduler (as the one we propose in this thesis). Even if the scheduler fails and the consumers have to resubmit the jobs, the resubmissions would happen less often than in the case of the provider's failure in the fully decentralized model, requiring less energy from the consumers.

A variant of this architecture is to make the scheduler use broadcast messages to retrieve the status information from the providers, reducing the network usage and eliminating the need to keep checking if the providers are still alive. However, the single point of failure is not eradicated.

4.1.3 Hybrid Architecture

In this architecture, we also have a central scheduler, but in this case, the consumers announce the jobs to the providers via broadcast messages. The providers are then responsible for performing the matchmaking process and interact with the scheduler to get the final decision about the job execution, avoiding further processing by the consumer.

Figure 12 shows an example of how this architecture may work.

- The steps 1 and 2 represent the registration of the job with the scheduler;
- In the 3, the consumer sends the job description to the providers;
- In steps 4 and 5, the providers tell the scheduler that they can process the job;
- In steps 6 and 7, the providers receive the decision of the scheduler;
- In steps 8 and 9, the chosen providers complete the task and inform the consumer and the scheduler, respectively.

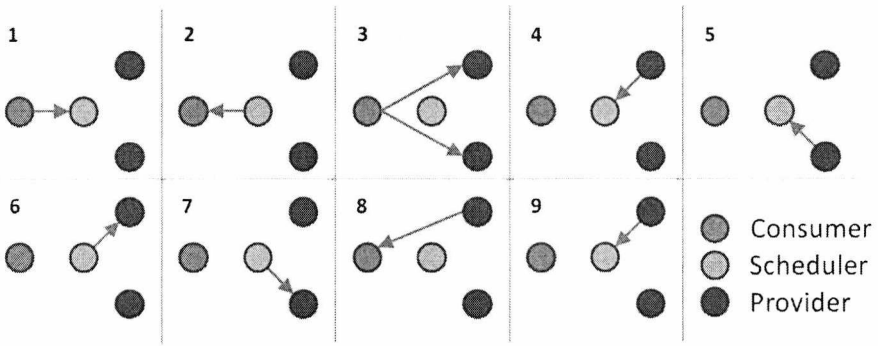

Figure 12 - Hybrid scheduling architecture.

Compared to the fully centralized architecture, the hybrid architecture seems to have the same number of messages exchanged. However, considering the presence of more providers, only those providers that can process the job will communicate with the scheduler, whereas in the fully centralized architecture there is always the communication between providers and schedulers (if such communication does not happen, the scheduler will end up with out-of-date information about the status of the providers to run the matchmaking process).

This architecture uses more network than the fully decentralized one, but it requires less processing by the consumers, which makes a better alternative for our solution based on the requirements specified in Chapter 2.

Considering the flow of information in Figure 12 and the reasoning presented in the section 4.1.1, we can use the following formula to compute the amount of energy spent by a consumer to submit a job:

$$Pjh = Ct + Cr + Cj.$$

Using this formula, we can estimate that a Nokia N95 device would spend approximately 168.12 mJ of energy to submit a job to the grid, independently of the number of providers available.

If the scheduler broadcasts the job description (step 3) instead of the consumer, the amount of energy required from the consumer to submit a job in the hybrid model becomes the same as in the fully centralized model (in other words, Pjh = Pjc).

In the case where providers fail, the analysis is also similar to the one presented for the fully centralized solution. We may have the case where the central device deals with those faults, freeing the consumer from further processing. However, in order to alleviate the process on the central scheduler (which we assume may be running in a limited device), the consumer may be in charge of some activities, such as sending reminders to the providers or the scheduler when a timeout for the job execution is reached, for example. Nevertheless, there is no need for the consumer to resubmit the job, unlike the fully centralized model.

Assuming that the consumer broadcasts the job description whenever a job is not completed in the expected amount of time due to a provider's failure, the energy cost for the consumer handling this kind of fault would be $M \times Cj$, where M is the number of failures. Therefore, considering the Nokia N95 as the consumer, it would spend 17.4 mJ (5×3.48) for $M = 5$.

Table 3 presents the amount of power that would be required from the Nokia N95 when running a face recognition application. For the calculations, we considered the example presented in the section 2.1.1, with a job of 409600 bits (50 KB). The table shows little difference between the power consumption for submitting a job (Pj), but much greater power consumption is required when using the fully decentralized model to resubmit jobs in case of faults of the providers Pr.

Table 3 - Power required from the N95 considering a face recognition application.

Model	Pj (N=10)	Pr (M=5)
Fully Decentralized	Pjd = 2059.48 mJ	Prd = 10240.00 mJ
Fully Centralized	Pjc = 2048.80 mJ	Prc = 0 mJ
Hybrid	Pjh = 2052.28 mJ	Prh = 17.4 mJ

Even though we still have a single point of failure in the hybrid solution, the fact that providers know about the submitted jobs allows us to develop fault tolerant mechanisms to reconstruct the status of the scheduler in case of failures without adding a higher load to the consumers (which would satisfy the requirement C.1 in the section 2.5), or requiring additional resources, such as a replicated server for the scheduler, for example.

Note that the fully centralized model may require the resubmission of the job when the scheduler fails and another one has to be selected, in which case we would have Prc = Pjc = 2048.80 mJ, making it much higher than Prh.

There can be many ways of organizing the communication between the devices in the grids in any of the presented models. However, we are mainly interested in providing a solution that does not add too much energy cost to the consumers and in finding a solution that can perform well having a central device running in a limited device.

The fully centralized or decentralized models may still be the best option for some types of applications and different environments, especially where energy and resources are not very constrained. In this thesis, we intend to make a generic system that can accommodate different applications.

For these reasons, we have chosen the hybrid model (which would satisfy the requirements A.1 and B.1 in the section 2.5), and the rest of this chapter describes our scheduling solution adopting the hybrid architecture, including the fault tolerance mechanisms.

4.2 Component-Based Solution

In our proposed solution, every device participating in the grid has a main piece of software composed by smaller components, which are described in this section.

Figure 13 shows a block diagram that represents the internal view of the main software. Each block represents a software component that handles certain functionalities. The details of how the functionalities should be implemented are presented in the remainder sections of this chapter.

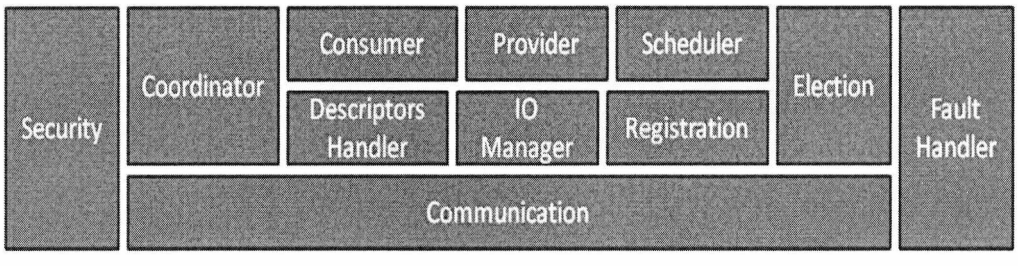

Figure 13 - Software components for each component in the proposed home grid.

With this modularity of the component's architecture, we can use different mechanisms for implementing each one of them, evaluate them separately and try to find the best solutions.

Although all these components are present in each device, they may or may not all be active at the same time, which means that a device can have only the scheduler active or be acting exclusively as a consumer, for example.

Communication

The communication component is responsible for sending and receiving messages in the network, including the mechanism for broadcasting information that is needed in our solution.

Registration

This component's duty is to register a device in the grid. The registering service has the naming service embedded in it, which gives the unique identification for the component/device in the grid system.

This is the component that finds the location of the scheduler and keeps this information to be used by the other components.

Consumer

This component is responsible for controlling the submission of jobs and making sure that they are completed, either locally or by a provider in the grid, depending on the policy of the application, as described later in this chapter.

Provider

This component is responsible for performing the matchmaking, communicating with the scheduler and controlling the execution of the jobs that arrive in the device. This component will not be active in those devices that are set as "consumer only", i.e., that can only submit jobs to the grid and do not process any job.

Scheduler

This component performs the following activities: issuing global job IDs, registering devices, replying to requests from providers and consumers, and carrying out the recovery process.

Election

This component is responsible for dealing with the processes involved in the election of a new scheduler, including the computation of the utility value and its distribution.

Descriptor Handler

This component provides mechanisms to map jobs and resource descriptors into a pre-defined standard. In this thesis, the standard adopted is the object-oriented scheme for the job and resource description presented in Appendix B (requirement D.3 in section 2.5).

IO Manager

Jobs may have some related input and output data. The location of this data is specified in the job descriptor, and can be a local or remote file or a database, for example. The component I/O Manager is the one responsible for reading and writing the data related to the jobs.

Coordinator

This component has a very important role. In our proposed distributed solution for job scheduling, the Coordinator is responsible for the communication between local components and for gathering information such as CPU load, the battery and memory status, which can be used by the other components.

Security

This component is in charge of assuring that the security policies for each resource are followed. The other duties for this component are access control to the grid, the privacy and integrity of data when required by the application submitting the jobs and the authentication of the other components.

The definition of the security techniques to be employed in the system was left as future work, due to time constraints.

Fault Handler

If any fault happens (e.g. network timeouts, application errors), the Fault Handler is the component that will be responsible for acting on the error or informing the appropriate component to handle it.

4.3 Device Registering and Resource Discovery

To avoid ambiguity about resources, we first need to define a mechanism to provide unique IDs. For this problem, Czajkowiski et al. [78] suggests two approaches: one is a naming service based on a naming authority; and the other is based on probabilistic techniques.

The first approach consists of a centralized entity responsible for issuing credentials for all resources, which can be of simple implementation, but it has the disadvantage of requiring a new component in the system that needs to be managed and maintained, and it may not be possible to guarantee an infrastructure for such a component in the home environment.

Globally Unique Identifier (GUID) [79] systems are examples of the second approach; they can provide random identifiers based on some information from the resources. GUID systems do not guarantee a unique identifier, but the probability of two devices having the same identifier is very low.

Based on the statements above, we proposed the use of GUIDs in this thesis, where each device generates its own. The main purpose of an identification in this thesis is for the generation of global IDs for jobs, where the scheduler adds its own ID (GUID) to it (explained later in this section).

When a device joins the grid, its registration component broadcasts a REGISTRATION_REQUEST message (containing its GUID) to the other components in the grid, and only the scheduler replies with a REGISTRATION_RESPONSE message, so that the new device can discover the location of this shared space. The entering device then keeps this information for future use.

In order to assure that there will not be repeated IDs, even though the probability is very low, the scheduler keeps a data structure with the received IDs and the IP address of the sender, and every time the scheduler receives a new one during the registration process, it checks if the ID already exists. If it does not exist, the scheduler adds it to the data structure, or updates the ID for the same IP address of the sender (if it has been registered before). Otherwise, if the ID does exist and the sender is the same as the one stored in the data structure (i.e., same IP addresses), the registration proceeds normally (the REGISTRATION_RESPONSE is sent by the scheduler); but if the IP addresses are different, the scheduler sends an EXISTING_ID message to the registering device. This indicates that the latter has to generate another GUID and restart the registration process. The workflow in Figure 14 summarizes this registration process.

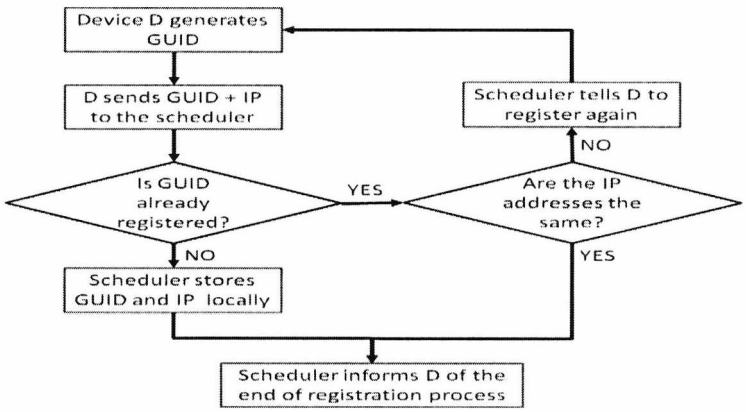

Figure 14 - Registration workflow.

Once the registering process is finished, the device can consume and/or provide resources in the grid. Furthermore, with this registering mechanism, the scheduler does not need to keep contacting the registered devices to check if they are active (requirement A.2 in section 2.5).

To submit a job to the grid, the consumer component submits a JOB_SUBMISSION_REQUEST message to the scheduler, which includes the description of the minimum resource requirement for the job execution, the input and output data (or an indication of where to locate them); and all the other information necessary to run the job.

In Figure 15, we present a pseudo-code for this step of the protocol. The *sendJob* procedure is invoked for each job of the application using the grid, and it assumes that the same consumer sends no other job at the same time, to improve fairness of the grid usage.

```
1. sendJob(job) {
2.      //the send procedure receives the type of message,
3.      //the destination, and the additional information
4.      send(JOB_SUBMISSION_REQUEST, scheduler, job);
5.      //lastJobSent stores the last job sent by the consumer
6.      lastJobSent = job;
7.      //starts the timer to wait for the response
8.      //from the scheduler before retrying sending the job again
9.      startTimer();
10.   }
```

Figure 15 - Pseudo-code for the consumers sending a job.

After receiving the JOB_SUBMISSION_REQUEST message, the scheduler stores the received information, generates a global ID for the received job, and sends a JOB_SUBMISSION_RESPONSE message (with this ID) to the consumer.

Figure 16 shows a pseudo-code that implements the behaviour of the scheduler when handling the job submission by the consumer. After executing this piece of code, the new job should have a global ID assigned and it should be stored in the scheduler's data structure, and the new global ID should start being sent to the consumer who owns the job. (Note that in the pseudo-codes in this thesis, the global ID generated by the scheduler is referred as GLOBAL_ID, while the LOCAL_ID refers to an ID of the job set by the consumer itself when creating the job).

```
1. handleJobSubmissionRequest(job) {
2.     //generates and sets the the global ID of the job
3.     job.GLOBAL_ID = generateID();
4.     //sets the job as available
5.     changeStatus(job, AVAILABLE);
6.     //add jobs to the data structure with jobs to be executed
7.     pendingJobs.add(job);
8.     //sends the response to the consumer with the global id
9.     send(JOB_SUBMISSION_RESPONSE, job.owner, job.GLOBAL_ID,
10.           job.LOCAL_ID);
11.   }
```

Figure 16 - Pseudo-code for the scheduler registering a job.

For this registering mechanism, we assume that the IP address is allocated manually or using the DHCP (Dynamic Host Configuration Protocol) [80], which is a protocol where a client requests an IP addresses to a server. Depending on the implementation, a DHCP server can allocate IPs to clients using one of following approaches:

- Dynamic allocation: the administrator defines a range of IP addresses that can be allocated to a client. The IP addresses are assigned to clients for a pre-defined period (lease time) that can be renewed/extended by the clients; this allows the DHCP server to reclaim (and then reassign those IPs) that did not have the lease renewed;
- Automatic allocation: The DHCP server permanently assigns a free IP address to a requesting client from the range defined by the administrator. This is like dynamic allocation, but the DHCP server keeps a table of past IP address assignments, so that it can preferentially assign to a client the same IP address that the client previously had;
- Static allocation: The DHCP server allocates an IP address based on a table with MAC address/IP address pairs, which are manually filled in (perhaps by a network administrator).

Home networks using wireless routers usually adopt DHCP with automatic or static allocation, which allow the clients to keep their IP addresses for a long period.

Our registering mechanism may not function well in those situations where the IP address of the clients changes frequently (e.g. DHCP server configured to use dynamic allocation with a short lease time), since the registering data structure may grow considerably (which may affect the performance of the registering service) and

it may eventually become inconsistent (an ID being associated with the wrong IP address). A better registering process is required for such situations (perhaps defining times for the scheduler to check the correctness of the values in the registering data structure, eliminating those that are no longer correct).

Since we do not consider the use of timestamps (as mentioned later in this thesis when talking about the scheduling system) – which also keeps the protocol simple as a response to the requirement D.1 in section 2.5, we assume that the job global ID is composed by a numeric value and the ID of the scheduler that assigned it. The numerical part of the global ID is used to indicate the order of arrival of the jobs at the scheduler, which is also the order for the execution of the job.

The ID of the scheduler is also part of the global ID in order to distinguish between two jobs with the same numerical global ID part, but issued by different schedulers, which can happen in the case where a network is partitioned, for example. We discuss more about the global ID later in the section on the fault tolerance of our mechanism.

After receiving the global ID from the scheduler, the consumer then broadcasts a JOB_DESCRIPTION message with the global job ID and the resource requirements to the other components, so that providers can compare these pieces of information with their own resource description, and verify if they are capable of executing that job (*matchmaking*).

In this way, we distribute the task of discovering resources between the components and not only as a task for the scheduler, which cannot be guaranteed to be powerful enough to perform the matchmaking and the scheduling itself without compromising the performance of the system. As shown later in section 5.4.6, for example, the matchmaking represents approximately 65% of the scheduling process in the OLB scheduler (increasing the percentage with the availability of more providers).

The Figure 17 shows a pseudo-code for the consumer's behaviour after it receives the job registration response from the scheduler with the global ID for the last submitted job. After the execution of this code, the consumer will have assigned the received ID to the submitted job, keep track of its submission, broadcast its description and start the timer for the job execution (explained later in this thesis).

In Figure 18, we have the pseudo-code for the provider's behaviour when it receives the job description broadcast by a consumer. The lines 2 and 3 of Figure 18

represent the provider requesting its own status to the local coordinator component and comparing it with the job requirements, respectively. The lines 4 to 8 implement the fault tolerant behaviour described later in this thesis in the section 4.5.4. The lines 10 to 16 implement the classification of the job according to the capability of the provider of executing it. The job is then stored in a local data structure (line 17) and a scheduling request is sent to the scheduler if the provider is not awaiting a response from a previous request (expression "*jobSent IS FALSE*" in the line 23).

```
1. handleJobSubmissionResponse(jobLocalID, jobGlobalID) {
2.     //checks if the response received is for the
3.     //last job sent by the consumer
4.     if (jobLocalID IS lastJobSent.LOCAL_ID) {
5.         if (lastJobSent.status IS NOT COMPLETE) {
6.             //sets the global ID on the consumer side
7.             lastJobSent.GLOBAL_ID = jobGlobalID;
8.             //marks the job as sent; in this example, by
9.             //by adding it to an internal list
10.            jobsSent.add(lastJobSent);
11.            //broadcasts the job description
12.            broadcast(JOB_DESCRIPTION,
13.                    lastJobSent.description);
14.            //start the timer for the job timeout
15.            startCompletionTimer(lastJobSent);
16.        } else {
17.            send(CANCEL_JOB, jobGlobalID);
18.        }
19.     } else {
20.         send(CANCEL_JOB, jobGlobalID);
21.     }
22.   }
```

Figure 17 - Pseudo-code for the consumers handling the jobs submission response.

```
1. handleJobDescription(job) {
2.     //retrieves the local status information for the matchmake
3.     local_status = coordinator.getStatus();
4.     //matchmake the status with the job requirements
5.     processable = matchmake(local_status, job.DESCRIPTION);
6.     if (isProcessingJob AND job IS chosenJob) {
7.         if (processable IS FALSE) {
8.             stopExecutingJob();
9.             send(CHANGE_STATUS, scheduler, job.GLOBAL_ID);
10.        }
11.    } else {
12.        if (processable IS TRUE) {
13.            classifyAsProcesable(job);
14.        } else {
15.            classifyAsNotProcesable(job);
16.        }
17.        jobs.add(job);
18.        requestSchedule();
19.    }
20.  }
21.
22.  requestSchedule() {
23.      if(hasProcessableJobs() && jobsSent IS FALSE) {
24.          queue = getFirstMProcessableJobs();
25.          off_interval = getOldestOffInterval();
26.          send(SCHEDULE_REQUEST, scheduler,
27.              queue, off_interval);
28.          jobsSent = TRUE;
29.      }
30.  }
```

Figure 18 - Pseudo-code for the providers handling the job description broadcast by consumers.

The information held by providers after the matchmaking process is also useful to reconstruct the scheduler's status in the case of failure of the scheduler, which is part of our proposed solution for fault tolerance.

The only central point in our grid is the scheduler component, which is simple to use and requires space for storage (which is cheap), but it does not need much processing power. These characteristics allow us to use devices that have limited processing capabilities to play this role.

This distributed resource discovery mechanism makes the limited devices in the grid perform some processing when they receive the broadcast messages, but it is preferable that they make a few comparisons and verify that they cannot execute the jobs and stop processing instead of concentrating the resource discovery task in only one central limited device.

These limited devices can also be set as "consumer only", which indicates that they cannot receive jobs to execute and their local components can then ignore the JOB_DESCRIPTION messages. This decision about setting a device as "consumer

only" can be done by the user, or a set of minimum requirements can be defined in order to enable the provider component in a device.

4.4 Scheduling System

So far, we have described how a device registers within the grid, and how consumers submit their jobs using broadcast messages to find suitable resources to execute them. In this section, we describe the behaviour of the grid participants: consumers, providers and scheduler. This also includes the messages that are exchanged between them.

For this specification, we make the following assumptions:

- The jobs sent by consumers are independent, so that they can all be executed in parallel (requirement D.2 in section 2.5);
- If the application jobs are dependent, the consumer sending them is responsible for sending them in the appropriate order, and delaying submissions until previous results that are needed have been received;
- Timestamps are not considered for scheduling purposes, in order to avoid additional processing and communication for clock synchronization in such a dynamic environment;
- A reliable communication protocol (e.g. TCP) is used for the transmission of unicast messages.

Note that, in this section, we only describe how the system works assuming that we have a reliable device (one that will never disconnect or fail) running the scheduler. The behaviour of the system with failures of the scheduler will be described later in this thesis.

After the *matchmaking* (see previous section), providers classify the job as *processable* or *not processable*, and then store it locally in a queue, following the order of the global ID. The storage of jobs that cannot be processed by a provider will be useful to reconstruct the status of the scheduler if it fails. During the scheduling process, providers send a SCHEDULE_REQUEST message to the scheduler with a queue (Pq) containing the first N jobs stored in their local queue that are tagged as *processable* (please refer to the Figure 18 for the pseudo-code that implements this behaviour). The scheduler receives this queue, selects the job that

should be processed next by the provider (the first job in Pq that has not been allocated yet), and updates the queue by removing those jobs that have already been completed (allocated jobs are kept in the queue so they can be rescheduled in case of failures). This updated version of the queue and the selected job are sent back in a SCHEDULE_RESPONSE message to the original provider; if the scheduler holds any data required by the allocated job (executable file and I/O data), that data is also attached to the SCHEDULE_RESPONSE message.

Providers may leave the grid at any time and there might be a number of jobs that have been broadcast while they have been unavailable, as illustrated in Figure 19.

In Figure 19, Job A represents the last job received by the provider while it was on for the first time; Job B represents the first job received when it became on again after the first off period. Jobs C and D are, respectively, the last and the first job received before and after the second off period, and so on.

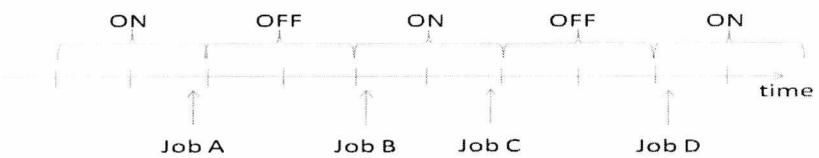

Figure 19 - Example of a job submission timeline where a provider becomes on and off.

In order to acquire those jobs that might have been broadcast during their off period, providers also inform the scheduler (within the SCHEDULE_REQUEST message) the oldest interval of jobs where there can be jobs missing (e.g. the global IDs of jobs A and B in Figure 19). Based on this information, the scheduler is able to find out if there are jobs to be processed that were sent during that particular off period; if there are any, the scheduler creates a list (Ls) with the first M jobs that have been received in the period, and sends it to the provider. The scheduler also informs the provider if there are more jobs in the current interval other than those M jobs present in the list (using here what is called the "more flag"). Both Ls and "more flag" are part of the SCHEDULE_RESPONSE message.

If there are no more jobs in the current interval of jobs, providers can start sending the details of the next interval (if there is one).

When providers receive the list of missing jobs from the scheduler, they perform the matchmaking for all jobs in the list and classify them as *processable* or not. The jobs are then placed in the local queue in order of their global IDs.

In Figure 20, we illustrate the messages that should be exchanged between providers and scheduler during the scheduling process.

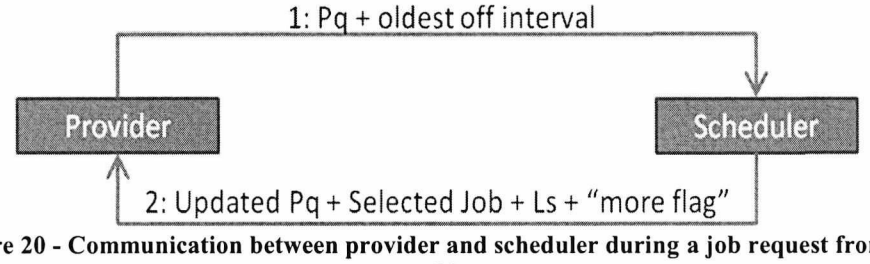

Figure 20 - Communication between provider and scheduler during a job request from the provider.

The procedure *scheduleRequest* in Figure 18 (lines 22 to 30) shows the pseudo-code for a provider sending the schedule request to the scheduler, corresponding to the step 1 in the Figure 20.

In the Figure 21, we show a pseudo-code that implements the behaviour of the scheduler when it receives the schedule request from a provider. The requests are queued and dealt individually by the order of arrival. The *schedule* procedure called in the line 2 represents the decision making step where a job is chosen based on the list of jobs sent by the provider. Its pseudo-code is presented later in this thesis in the Figure 47. The rest of the code in Figure 21 implements retrieval of any missing jobs (jobs submitted while the provider was unavailable), if any, and the submission of the schedule response to the provider.

```
1. handleScheduleRequest(queue, off_interval) {
2.     //choose the job to be executed from the received queue
3.     chosenJob = schedule(queue);
4.     jobsList = createEmptyList();
5.     //populates the jobsList with missing jobs and returns
6.     //TRUE if there are more missing jobs, or FALSE otherwise.
7.     more_flag = getMissingJobs(jobsList);
8.     //send the schedule response to the provider
9.     send(SCHEDULE_RESPONSE, chosenJob, queue,
10.           jobsList, more_flag);
11.  }
```

Figure 21 - Pseudo-code for the scheduler handling the schedule request from providers.

The pseudo-code showed in the Figure 22 implements the behaviour of a provider handling the schedule response from the scheduler. In the lines 4 to 14, the

provider performs the matchmake on the missing jobs that it has received from the scheduler (if any), and then updates its local jobs queue.

When a provider finishes the execution of a job, it sends the result to the location indicated in the job description (e.g. directly to the job's owner or to a shared repository) to avoid extra communication. The message containing the result is called JOB_COMPLETION_CONSUMER. Once that completion message is sent, another message, here identified as JOB_COMPLETION_SCHEDULER, is immediately sent to the scheduler to inform it that the job has been finished, so it can be removed from the local data structure.

```
1. handleScheduleResponse(chosen_job, updated_queue,
2.         missing_jobs, more_flag) {
3.     //matchmake the missing jobs
4.     for (i = 0 .. missing_job.size) { xxxx
5.         job = missing_jobs[i];
6.         processable = matchmake(local_status,
7.             job.DESCRIPTION);
8.         if (processable IS TRUE) {
9.             classifyAsProcesable(job);
10.        } else {
11.            classifyAsNotProcesable(job);
12.        }
13.    }
14.    updateLocalQueue(updated_queue, missing_jobs); xxx
15.    jobsSent = FALSE;
16.    if (more_flag IS FALSE) {
17.        removeOldestOffInterval();
18.    }
19.    if (chosen_job IS NOT NULL) {
20.        result = execute(chosen_job);
21.        //writes the result to the specified location
22.        writeResult(result, chosen_job.outputLocation);
23.        //informs the consumer of the job completion
24.        send(JOB_COMPLETION_CONSUMER, chosen_job.owner);
25.        //informs the scheduler of the job completion
26.        send(JOB_COMPLETION_SCHEDULER, scheduler,
27.            chosen_job.GLOBAL_ID);
28.        completedJobs.add(chosen_job);
29.    }
30.    requestSchedule();
31.  }
```

Figure 22 - Pseudo-code for the provider handling the schedule response from the scheduler.

After sending the completion message to the scheduler, providers must keep the information about the completed job for some time. This information will be used for the recovery of the scheduler after a fault (explained later in this thesis).

These steps are illustrated in the lines 19 to 29 of Figure 22.

Figure 23 illustrates the complete flow of information for the proposed job scheduling mechanism.

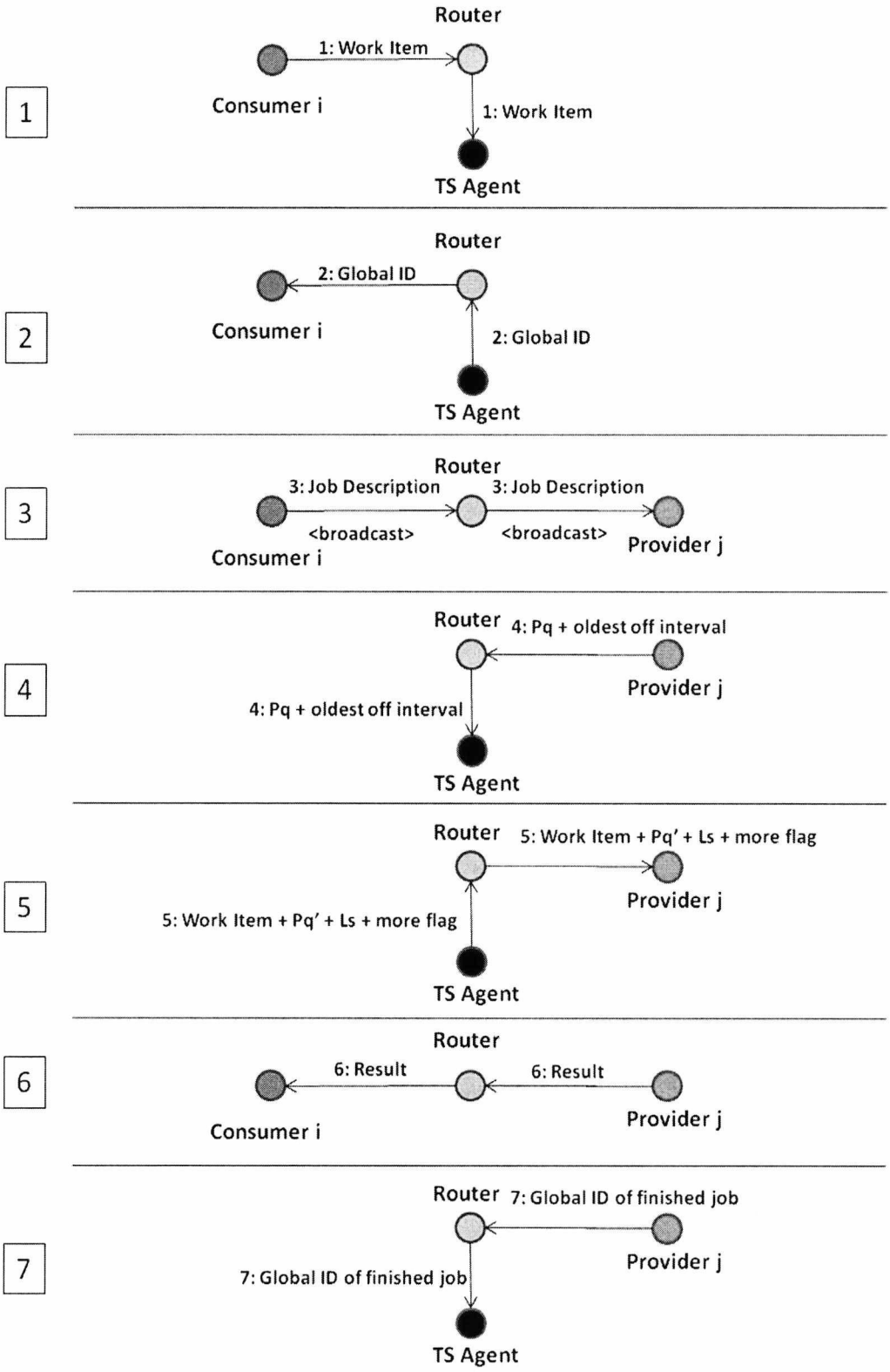

Figure 23 - Scheduling information flow.

4.5 Fault Tolerance Mechanisms

In this section, we describe how components in the grid should act when a failure is identified, in order to recover the system from such a failure.

Here we assume that tasks submitted to a home grid are not going to take a very long time to run, in contrast to the tasks in conventional Grids. For this reason, we are not considering the support of checkpoints, which would add more traffic to the network (which can be a bottleneck to the system as shown in the simulation results in Chapter 5), and would require a reliable device to act as the checkpoint server.

Since we are assuming that the device that is kept on most of the time has low capabilities and it is going to be running the scheduler, we do not want to add more processing to this device (e.g. playing the role of checkpoint server as well) and so have the performance of the Grid system degraded.

Since it is already difficult to guarantee at least one device will be kept on most of the time to run the scheduler, we cannot adopt replication of the scheduler to recover its status when it crashes. In this case, we take advantage of our P2P infrastructure to elect a new scheduler and to recover the queue of tasks.

Before we describe the fault handling mechanisms for crashes of the scheduler, providers and consumers, we need to specify how failures are detected.

4.5.1 Failure Detection

Because of the low capabilities of the scheduler and because of the negative impact on the performance of the Grid caused by network congestion mentioned earlier, we want to keep the failure detection mechanism as simple as possible, avoiding the extra processing that would be needed at the scheduler to manage "*are you alive?*" messages, which also implies more network usage. For these reasons, we adopt timeouts as the methods for detecting faults.

Network timeouts are used to indicate a node crash or a network failure.

Scheduling a task in our system depends on providers receiving the broadcast message sent by consumers. Since this broadcast is done using UDP, there is no guarantee that providers will receive the message. It may also happen that there is no provider online when the message is sent, but immediately after it is sent, a provider might become on and will not have received the message. For these reasons, consumers set up a timeout for the processing task to be completed and, if this

timeout is reached before the task is completed, the consumer can take further action (e.g. broadcast the message again or remove the task from the scheduler).

After sending a JOB_SUBMISSION_REQUEST message, consumers also set up a timeout for the response from the scheduler (see pseudo-code in Figure 15). If the timeout is reached before the consumer receives a response from the scheduler, consumers can resubmit the job or choose to process it themselves. If the consumer receives the JOB_SUBMISSION_RESPONSE message after processing the job itself, the consumer sends to the scheduler a CANCEL_JOB message containing the job ID to inform the completion of the job and allow the scheduler to update its own local queue (see pseudo-code in Figure 17).

The pseudo-code in Figure 24 implements how the consumers behave when they do not receive a response from the scheduler before the timeout. Note that we assume that the job registration is not synchronous, so the timeout handling in Figure 24 is not related to a connection/network timeout, which is used to identify a scheduler's failure, as explained in the next section.

```
1. handleJobSubmissionTimeout() {
2.     //read application policy to
3.     //decide what to do: process the job locally
4.     //or resubmit the job
5.     if (POLICY IS RESUBMIT) {
6.         send(lastJobSent);
7.     } else if (POLICY IS EXECUTE) {
8.         execute(lastJobSent);
9.         lastJobSent.status = COMPLETE;
10.           jobsCompleted.add(job);
11.    }
12.  }
```

Figure 24 - Pseudo-code for the consumer handling the job submission timeout.

4.5.2 Scheduler Crash

When a device gets a connection timeout while sending a message to the scheduler, it means that the scheduler has become unavailable, and the system might need to be recovered. The device that identified the problem then broadcasts a message to inform the others about the failure.

As explained earlier, we are not considering the replication of the scheduler. Therefore, we have to find a new device to play the role as a scheduler, which is

done via an election by the devices currently connected to the grid. This election is similar to the leader election problem in distributed systems.

Leader Election Algorithms

Le Lann [81] proposed the first algorithm to be applied to networks with a unidirectional ring topology [82], which is not the case of home grids. The solution assumes that each node has a unique ID, and that ID is sent around the ring. The node with the highest/lowest ID becomes the leader. If our solution was based solely on the ID, there could be the possibility of a fully loaded device being elected as the new scheduler, and this would degrade the performance considerably.

There is a broad range of solutions for the leader election problem, and they vary according to the communication mechanism (asynchronous vs. synchronous), the name of the processes/nodes (unique identities vs. anonymous), and the network topology (e.g. ring, tree, complete graph).

Election algorithms for synchronous communication are not suitable for the purpose of this thesis, since we assume that the devices do not know the number of other connected devices or their locations. The election in our proposed system is not random, nor based on the identity of the nodes; it is based on a set of variable that define the value of the node to the system. We also do not assume a static topology for the home network. Based on these features, the closest related leader election algorithm to the one proposed here are the ones used to elect coordinators in ad-hoc networks.

Vasudevan et al. [83] presents a short survey on election algorithms and a solution for the problem in ad-hoc networks. Most of the algorithms studied assume a static topology (e.g. [84] and [85] that assume a ring topology where nodes are always connected during the election), a topology does not change during the election process (either before the election start or after the new leader is elected), or an unrealistic communication model such as a network that preserves the order of the messages. Such assumptions do not apply to a home environment network.

The algorithm proposed in [83] chooses the *most-valued-node* to be the new leader/coordinator, instead of a random or ID based selection. Such valuation is based on some characteristic such as the maximum remaining battery life, or the node with minimum average distance from the remaining nodes. The proposed solution is based on a spanning tree [86] for the diffusion of messages and is tolerant

to dynamic topologies. It requires the participants to forward election messages and to send back some protocol-specific acknowledgment messages (*ack*). The evaluation of the algorithm shows that the use of broadcast messages produced better results than the use of TCP and UDP for node-to-node communication.

The election algorithm proposed in this thesis uses broadcast messages and does not require *ack* messages from the participants, saving bandwidth.

According to Kim et al. [87], we can distinguish two strategies for how systems adapt to failures. The first strategy consists of suspending temporarily the execution of the system in order to reorganize it, with components of the system evaluating themselves and competing with each other for the leadership. The second strategy is to allow the system to continue its correct operation and recover the system from the failure (which represents a very complex mechanism). The paper then presents a leader election algorithm with complexity between the two approaches mentioned before, where the leader is replaced with a pre-elected leader based on performance, links and connectivity. When this replacement happens, the first node that identified the leader's failure makes the final decision on the new optimal pre-elected leader. The problem with the algorithm is that it does not specify any behaviour for the cases where more than one node identifies the failure at the same time.

Our solution follows the approach where the activities are suspended for a short period, since it is simpler and because of the limited infrastructure to allow pre-elections of a new leader (scheduler).

Utility Function

The scheduler election process in this thesis is based on the concept of node utility presented in [88]. The utility u of a node, in our case, represents the capability of a certain device to run the scheduler, i.e., its value to the grid.

In [88], the utility value is used to elect the player that will coordinate a virtual region in a MMOG (Massively Multiplayer Online Game) built on a P2P network, and is computed by considering 6 factors that could influence the behaviour of the system: average latency distance, available bandwidth, player's reliability, available CPU, available memory and duration time of player in some virtual region. All these values are collected from all peers by a device responsible for coordinating the election, and placed in a matrix $X = (x_{ij})_{nm}$, where n is the number of peers and m is the number of factors.

Because the value of some factor may be contrary to utility, the values in X are normalized according to the following rules (consider u_{ij} the normalized value of x_{ij}):

1) In case better utility is achieved with a bigger value of a certain factor:

$$u_{ij}=\frac{x_{ij}-x_j^{min}}{x_j^{max}-x_j^{min}}$$

Where $x_j^{max}=\max_{1\le i\le n}\{x_{ij}\}$ and $x_j^{min}=\min_{1\le i\le n}\{x_{ij}\}$.

2) In case better utility is achieved with a smaller value of a certain factor:

$$u_{ij}=\frac{x_j^{max}-x_{ij}}{x_j^{max}-x_j^{min}}$$

3) In case better utility is achieved with the value of a certain factor being kept around x*:

$$u_{ij}=\begin{cases}\dfrac{x_{ij}-x_j^{min}}{x^*-x_j^{min}}, & x_{ij},\ x_j^{min}\le x_{ij}\le x^*\\[2ex] \dfrac{x_j^{max}-x_{ij}}{x_j^{max}-x^*}, & x_{ij},\ x^*\le x_{ij}\le x_j^{max}\end{cases}$$

The normalized matrix of X, denoted as matrix $U=(u_{ij})_{nm}$, is then used to elect the new coordinator. Some factors can also be prioritized by applying some different weights to them.

As we can see, the election mechanism adopted in [88] takes into consideration the maximum and minimum values of all the factors for all peers, and concentrates the computations of utility in a single device. In case of failure of this device, another one (being used as back-up) takes over the election process.

In order to use the same election mechanism described above to elect the new scheduler, we would need to specify another method to choose the election coordinator, and the behaviour of the system in case of failure of this coordinator. To avoid this extra processing, we have adapted the method above, based on the characteristics of our system.

But first, we have to define the set of influencing factor ($F = \{f_j\}$, $j = 1, 2, \ldots, m$) for our grid system. For this thesis, we consider the following 5 ($m = 5$) factors:

- Power supply (f_1): indicates the power level of the device in terms of percentage; if the device is not running on battery, it indicates 100% of power; otherwise, it considers the level of the battery divided by 2, so that a device on AC power has higher utility value than a device running on battery power with 100% of capacity. In this case, the higher is the percentage, the better is the utility, since we want a device that is on for most of the time as the scheduler;
- Storage (f_2): shows the storage capabilities of the devices. It is important that the new device to run the scheduler has enough space to store the scheduling queue and eventual data that consumers might want to send to the scheduler. Due to the difficulty to define appropriate bounds to the storage factor without adding extra load to the network, we consider that:

$$f_2 = \frac{1}{capacity},$$

  where *capacity* is the amount of free space available for storage and is ≥ 1 MB. If *capacity* < 1, f_2 is set to 1, so that when we normalize the value of this function (see later in this section), its utility is set to 0 (zero), which reduces the overall utility value for the device, since we assume that higher storage contributes to a better utility;
- Availability (f_3): indicates how reliable the device is in terms of connection to the grid. Every device is required to record the time of the first connection (T_0) to the grid, as well as the total time the device remained connected to the grid (Tc). Considering Ct as the current time when a device computes the availability value, we assume:

$$f_3 = \frac{Tc}{Ct\text{-}T_0}$$

  High availability contributes to better utility value, since we do not want the scheduler running in a device that keeps disconnecting or crashing;
- Available CPU Power (f_4): describes how powerful (in terms of CPU) the device is and how much of its CPU is available. As presented later in Chapter 5, the performance of the system is better when the scheduler is running on powerful devices. For this reason, we consider that devices with higher available CPU power should have higher utility. Following the same reasoning used for f_2, we have:

$$f_4 = \frac{1}{available\_cpu\_frequency},$$

where *available_cpu_frequency* is the frequency of the available CPU power and is ≥ 1 MHz. If *available_cpu_frequency* < 1, f_4 is set to 1, so that when we normalize the value of this function (see later in this section), its utility is set to 0 (zero);

Considering the cases for normalization presented earlier in this section, and based on the description of the influencing factors above, we can say that f_1 and f_3 fall into the case (1) of normalization, while f_2 and f_4 follow case (2).

In this thesis, we want devices to be able to compute their own utility values, since we do not have a device coordinating the election, and we also do not have a central system that can provide information about the other devices (like in conventional grid systems) in order to generate the matrix X.

Therefore, we adopt pre-defined maximum and minimum values (see below). This way, assuming that x_j is the value of the factor f_j for the device, we have adopted the following rules for computing the normalized value u_j:

1) In case better utility is achieved with a bigger value of a certain factor:

$$u_j = \frac{x_j - min_j}{max_j - min_j}$$

Where max_j and min_j ($max_j > min_j$) are the pre-defined maximum and minimum values for f_j mentioned above.

2) In case better utility is achieved with a smaller value of a certain factor:

$$u_j = \frac{max_j - x_j}{max_j - min_j}$$

3) In case better utility is achieved with the value of a certain factor being kept around x*:

$$u_j = \begin{cases} \frac{x_j - min_j}{x^* - min_j}, & x_j,\ min_j \le x_j \le x^* \\ \frac{max_j - x_j}{max_j - x^*}, & x_j,\ x^* \le x_j \le max_j \end{cases}$$

Considering f_1, we could define a maximum power supply (max_1) of 100% (device AC power) and a minimum (min_1) of 0% (with 50% representing a device running on battery with 100% of power as specified earlier in this section).

As for f_j (j = 2, 3, 4), max_j is set to 1, while min_j is set to 0.

Once the normalized value is computed for all factors, the total utility u for the device is computed as follows:

$$u = \sum_{j=1}^{m} u_j$$

Weights (W = {w_j}, j = 1, 2, ..., m) can also be defined for each factor. In this case, we can have a generic formula to compute the total utility:

$$u = \sum_{j=1}^{m} w_j u_j,$$

where $0 \leq w_j \leq 1$ and $\sum_{j=1}^{m} w_j = 1$. Factors considered more relevant for the computation of the utility value should have higher weight.

Proposed Scheduler Election Algorithm

Once a device finds out that the scheduler has crashed, it computes its own utility value (devices set as not scheduler capable – NSC – have utility value of "-1") and then broadcasts it in a specific message (SCHEDULER_FAILURE) that indicates that the failure was found and the election process is being started. After broadcasting that message, scheduler capable (SC) devices set a timer for T seconds, while NSC devices set the timer for (2 x T) seconds.

The pseudo-code in Figure 25 implements the step above. The *startElection* procedure should be invoked when a consumer or provider fail to communicate with the scheduler. After the execution of this code, the scheduler failure message should be broadcast, the timer started and those devices that are scheduler capable should set themselves as a candidate.

These timers count the time that devices have to wait for the completion of the election process. If the election has not been completed when then timer reaches its timeout, the timer's owner restarts the election process by attaching its own utility value to a RESTART_ELECTION message that is also broadcast (the device is not required to compute the utility value again).

```
1. startElection() {
2.     startElection(SCHEDULER_FAILURE);
3. }
4.
5. startElection(message) {
6.     utility = computeUtility();
7.     broadcast(message, currentScheduler, utility);
8.     if (schedulerCapable IS TRUE) {
9.         timer.start(T);
10.        isCandidate = TRUE;
11.    } else {
12.        timer.start(2*T);
13.    }
14.  }
```

Figure 25 - Pseudo-code for the start of an election process.

The pseudo-code in Figure 26 shows an example of how the election timeout can be handled. This code uses the *startElection(message)* procedure in Figure 25.

```
1. handleElectionTimeout() {
2.     if (isCandidate IS TRUE) {
3.         currentScheduler = THIS;
4.         broadcast(SCHEDULER_ELECTED, utility, THIS);
5.         coordinator.startRecoveryTimer();
6.     } else {
7.         startElection(RESTART_ELECTION);
8.     }
9. }
```

Figure 26 - Pseudo-code for the election component handling the election timeout.

When a device receives the SCHEDULER_FAILURE message, it computes its own utility value ("-1" for NSC devices) and compares it to the one received with the message. If the computed utility is greater than the one received with the message, the device broadcasts its utility (HIGHER_UTILITY message) to indicate that it is now a candidate and starts a timer (*t1*) with T seconds. If the computed utility is less than or equal to the received one, the device stops any timer that have already been initiated, disregards itself as a candidate, then starts another timer (*t2*) with (2 x T) seconds, and awaits for the final decision about the new scheduler or until *t2* reaches the timeout, when it should restart the election.

An implementation for the steps above is presented as pseudo-code in Figure 27. The condition is the line 2 is to avoid the code between the lines 3 and 31 to be executed multiple times in the cases where more than one component identifies the scheduler's failure at the same time.

If the timer of the current candidate reaches the timeout (which means that it has not received any greater utility value than its own), it broadcasts another message (SCHEDULER_ELECTED) with its utility value attached, to announce itself as the new scheduler, and starts the process to recover the status of the job's queue (see the pseudo-code in Figure 26).

The reason for *t2* being longer than *t1* is to avoid more than one candidate declaring themselves as the scheduler at the same time.

```
1. handleSchedulerFailureMessage(received_utility) {
2.     if (electionStarted IS FALSE) {
3.         if (schedulerCapable IS TRUE) {
4.             local_utility = computeUtility();
5.             if (local_utility > received_utility) {
6.                 broadcast(HIGHER_UTILITY, local_utility);
7.                 timer.start(T);
8.                 isCandidate = TRUE;
9.             } else {
10.                isCandidate = FALSE;
11.                   if (timer.started IS TRUE) {
12.                       timer.stop();
13.                       timer.start(2*T);
14.                   }
15.                }
16.            } else {
17.                if (timer.started IS FALSE) {
18.                    timer.start(2*T);
19.                }
20.            }
21.            electionStarted = TRUE;
22.        }
23.    }
```

Figure 27 - Pseudo-code for the election component handling the SCHEDULE_FAILURE message.

Since we assume that there is no guarantee that the broadcast messages are delivered, it may also happen that some devices do not receive the announcement message from the new scheduler or even the HIGHER_UTILITY messages. Those devices may attempt to restart the election or, if there is a candidate that has not received the messages, announce another scheduler as elected. If any device that received the new scheduler's announcement receives a RESTART_ELECTION message or a SCHEDULER_ELECTED from a device with lower utility value (including the scheduler itself), it then broadcasts the ELECTION_COMPLETE message with the address and utility value of the elected scheduler. If a new

SCHEDULER_ELECTED message brings a higher utility value than the one from the previous SCHEDULER_ELECTED message, we have two possible behaviours:

- The message is received after the recovery process is complete: devices keep the active scheduler and inform the sender about it, so that the system can continue working;
- The message is received during the recovery process: the devices assume the sender as the new scheduler and start a new recovery process; they also inform the previous scheduler of the change, so that the device running it can also take part in the recovery process.

It may also be possible that, due to multiple losses of packets in the network, a group of devices do not receive the SCHEDULER_ELECTED message, and ends up electing another scheduler among themselves constituting a partition. When this happens, we use the same mechanism for detecting the presence of multiple schedulers and synchronizing them as after a network partition. This mechanism is explained later in this chapter.

The pseudo-code in Figure 28 shows the behaviour of the election component when it receives the SCHEDULER_ELECTED message. The condition in the line 3 is to check if the component has already received such message from the same scheduler before, in which case no further processing is required. The lines 5 to 17 implement the behaviour in the case where the device processing the message also runs the scheduler. Since we consider factors such as the connected time to the grid for the computation of the utility value, we expect a low probability of devices computing the same utility value. However, if that happens, the decision of the new scheduler is done randomly by adding a random number between 0 and 1 to the previous utility and broadcasting the result. This situation is implemented by the code in the lines 6 to 11. Eventually one of the devices competing for the scheduler role will end up with a higher utility value and be elected, being acknowledged by the losing devices, which will start the recovery process.

```
1. handleSchedulerElected(senderUtility, newScheduler) {
2.     timer.stop();
3.     electionStarted = FALSE;
4.     if (currentScheduler IS NOT newScheduler) {
5.         if (THIS is currentScheduler) {
6.             if (utility IS senderUtility) {
7.                 //generates a random number between
8.                 //0 and 1 and adds it to the utility
9.                 random = RANDOM(0, 1);
10.                utility = utility + random;
11.                broadcast(SCHEDULER_ELECTED, utility, THIS);
12.            } else if (utility > senderUtility) {
13.                broadcast(SCHEDULER_ELECTED, utility, THIS);
14.            } else {
15.                currentScheduler = newScheduler;
16.                coordinator.startRecovery();
17.            }
18.        } else {
19.            currentScheduler = newScheduler;
20.            coordinator.startRecovery();
21.        }
22.    }
23.  }
```

Figure 28 - Pseudo-code for the election component handling the scheduler elected message.

4.5.3 Recovering the Scheduler

Once the new scheduler has been selected, it is necessary to recover its status, which includes:

(i) Reconstructing the local queue of jobs;

(ii) Finding the correct identity to be assigned to the next job sent by a consumer.

After receiving the SCHEDULER_ELECTED message, active consumers send to the new scheduler the global ID of the last job accepted by the previous scheduler (RECOVER_ID message), while active providers send the queue with known jobs and the queue with the completed jobs (RECOVER_QUEUE message). The procedure *coordinator.startRecovery* called in the lines 16 and 20 in the Figure 28 represent this step. The coordinator component then invokes the corresponding procedures in the consumer (presented in the Figure 23) and provider (presented in the Figure 24) components.

```
1. startRecovery() {
2.     lastKnownID = jobsSent.lastJob.GLOBAL_ID;
3.     send(RECOVER_ID, scheduler, lastKnownID);
4. }
```

Figure 29 - Pseudo-code for the ID recovery performed by the consumers.

```
1. startRecovery() {
2.     send(RECOVER_QUEUE, scheduler, jobs, completedJobs);
3. }
```

Figure 30 - Pseudo-code for the queue recovery performed by the providers.

The scheduler then identifies the greatest global ID (referred to in this thesis as the recovered ID) from the IDs sent by consumers and from the ID of the last job in the queues sent by providers (note that the queues are ordered by the global ID). Jobs submitted after the recovery process will be assigned with a global ID greater than the recovered ID.

The scheduler's queue is recovered by putting together the jobs known by providers and removing those that have already been processed.

The pseudo-codes in Figure 24 show how the scheduler handles the recovery messages. For the *handleRecoverQueue* procedure, it is assumed that the *pendingJobs* and the *completedJobs* queues are ordered by the jobs global IDs. The *updatePedingJobs* and the *updateCompletedJobs* are not required to keep such order in the scheduler's local data structure, since the scheduling process is done based on the lists sent by the providers, which are expected to be ordered.

```
1. handleRecoverID(jobID) {
2.     if (jobID > lastIssuedID) {
3.         lastIssuedID = jobID;
4.     }
5. }
6.
7. handleRecoverQueue(pendingJobs, completedJobs) {
8.         handleRecoveredID(pendingJobs.lastJob);
9.         handleRecoveredID(completedJobs.lastJob);
10.        updatePendingJobs(pendingJobs);
11.        updateCompletedJobs(completedJobs);
12.    }
```

Figure 31 - Pseudo-code for the recovery performed by the scheduler.

When a device that was not active during the recovery process becomes active again, its registration component should receive the recovered ID from the new

scheduler. The recovered ID is then forwarded to the coordinator component that is responsible for communicating with any consumer or provider component to be activated on that device.

Consumers then have to resubmit all their pending jobs that have a new global ID greater than the recovered ID; whereas providers discard all the jobs in their queues that have a global ID greater than the recovered ID, since it is most likely that the job may have another global ID assigned, or that the consumer is not active to be informed of the change on the ID.

Providers also send the queue of completed jobs to the scheduler, so the latter can update its own local queue, completing the registration process of that provider, and allowing the returning provider to start sending the queues and ranges as part of the scheduling process.

After the recovery process, when the new scheduler receives an unknown job with global ID smaller than the recovered ID, two actions are possible:

a) The job is discarded if it is sent by a provider during the scheduling process;
b) The job is added to the scheduler's queue when it comes from a reminder message from a consumer.

At the beginning of the recovery process, the new scheduler starts a timer for a short amount of time T seconds that indicates how long the recovery process should last. After that time, the scheduler checks if there are still recovery messages to be processed; if that is the case, the timer is reset for another T seconds. This process continues until there are no more recovery messages to be processed when the last T seconds have elapsed, and the scheduler sends a RECOVERY_COMPLETE message, indicating that consumers and providers can be (re)activated.

The pseudo-code in Figure 32 shows how the scheduler handles the recovery timeout. It assumes that the timer for the recovery process has been started after the election and that the recovery messages that arrive are queued until they can be processed.

```
1. handleRecoveryTimeout() {
2.     if (hasMoreRecoveryMessages() IS TRUE) {
3.         timer.startRecoveryTimeout(T);
4.     } else {
5.         broadcast(RECOVERY_COMPLETE);
6.     }
7. }
```

Figure 32 - Pseudo-code for the scheduler handling the recovery timeout.

The main objective for this protocol is to allow a quick restart of the system with the recovery of most of the previous status of the scheduler, without much interaction with consumers, especially those running on battery.

4.5.4 Provider Crash

Providers and consumers are the nodes in our grid that are most likely to leave the Grid at any time. If the provider crashes immediately after sending its list of candidate jobs to the scheduler, the scheduler will not be able to contact that provider in order to deliver the information about the selected job and the updated list. When this happens, the scheduler simply changes back the status of the selected job to its initial "available" state, and waits for another provider to indicate that it is capable of processing that job.

If the provider stops working while processing a job, the rest of the system will continue to work. Eventually, the consumer reaches a timeout for that job and may choose to broadcast the job description again. In this case, once the scheduler receives the job again, it attempts to contact the provider that was selected to process the job in order to check if it is still working; if the communication with the provider fails, the scheduler changes the status of the job to "available", so it can be processed by another provider; otherwise, no further action is required from the scheduler.

It may be possible that the provider has not completed the job because its processor has been highly used by other applications. For this reason, once the provider identifies that the retrial message is for the same job it is processing, the provider re-evaluate the requirements for processing the job against its own resources to check if it is still able to carry on processing the job. If the provider finds out that it cannot match the required resources for executing the job (e.g. insufficient available CPU), the provider stops processing the job and sends a CHANGE_STATUS message (carrying the job global ID) to the scheduler to change

the status of the job to "available", so that another provider can process it. The pseudo-code that implements this behaviour is presented in Figure 18.

4.5.5 Consumer Crash

When a consumer sends a job to the grid, it specifies the description of the job, including where providers can get the execution code/file, and the I/O data. If the location of these is the consumer itself, it means that the consumer must stay on during the period of execution of the task, so the provider selected to execute the job can access the data.

If the consumer crashes before the provider is able to get all the necessary information to execute the job, the provider informs the scheduler of the failure (using the CONSUMER_UNAVAILABLE message, carrying the ID of the unavailable consumer), and all the jobs from that consumer are removed from the scheduler.

The pseudo-code in Figure 33 shows how the providers handle the connection timeouts for the messages they send. The lines 2 and 3 implement the case described above, where the provider informs the scheduler that a consumer has crashed. The lines 4 to 5 show that the provider starts an election if any of the messages exchanged with the scheduler fail to be delivered.

```
1. handleConnectionTimeout(message, job) {
2.     if (message IS JOB_COMPLETION_CONSUMER) {
3.         send(CONSUMER_UNAVAILABLE, scheduler, job.owner)
4.     } else if (message IS JOB_COMPLETION_SCHEDULER
5.             OR message IS SCHEDULE_REQUEST
6.             OR message IS CHANGE_STATUS
7.             OR message IS RECOVER_QUEUE
8.             OR message IS CONSUMER_UNAVAILABLE) {
9.         coordinator.startElection();
10.    }
11.  }
```

Figure 33 - Pseudo-code for the provider handling connection timeout.

Another option is to consider setting the jobs from the consumer as "suspended" and when the consumer becomes online again, during the registration process, the scheduler asks the consumer if the jobs must be set as "available" again, or removed. The problem with this approach is that the jobs may stay in the scheduler's local queue for a long time, just increasing its size and, consequently, increasing the need

for processing power to schedule a job, affecting negatively the performance of the system.

In this thesis (including the model checking in Chapter 7), we use the first option, which is simpler and involves less network usage.

4.5.6 Job Replication

In our proposed scheduling model, it is not possible to guarantee that a certain job will be executed only once. There might be a case where a provider (P1) executes a job and becomes unavailable before sending the result to the consumer and informing the scheduler that the job has been completed. If the same provider stays off for a period that is long enough for the consumer to detect a timeout for that job, another provider (P2) might be selected to execute this job.

Once online again, P1 then sends a JOB_PENDING message to the scheduler to indicate that the job has been executed, but with results pending submission to the consumer that owns the job. The scheduler then checks the status of the job. If the job had not been allocated to another provider, the scheduler would send a PROCEED message to P1, indicating that P1 can complete the job execution process. This is to avoid extra communication with the consumer, which may be running on battery power.

Considering the case where P1 executed a job and became unavailable before informing the consumer and the scheduler, and that the job has already been reallocated to P2, there are a number of strategies that can be adopted. Here we list four possibilities:

1. The scheduler sends a PROCEED message to P1 and allow both providers to send the results to the consumer without any concurrency control. In the case where one of the providers fails, the other one may be able to deliver the result of the job execution. This is the simplest solution and may not be the most efficient solution, especially because it may require more processing and communication from consumers.
2. The scheduler may send a HOLD message to P1 after verifying that P2 is still alive (using an "is alive?" message). If the HOLD message makes P1 wait for the completion of the job and not process any other job, there may be waste of processing power of P1 if the job is big and takes a long time to be executed. However, this option would avoid the need for

concurrency control as a result of both P1 and P2 trying to write the results to the same place at the same time. It may also happen that P2 crashes just after replying to the "is alive?" message sent by the scheduler, causing the job not to be completed by any of the providers before the next timeout for the job on the consumer. To avoid this situation, the scheduler can send periodic "is alive?" messages to P2, and whenever it finds that P2 has crashed, it sends a PROCEED message to P1. At this point, the scheduler will only keep checking if P1 is alive in the case where P2 comes back online (and it is then sent a HOLD). If P2 completes the job, the scheduler sends a CANCEL_PROCESS message to P1, allowing it to continue with the processing of other jobs. The use of the "is alive?" message increases the network usage and may considerably degrade the performance of the system if this situation repeats very often, and for many providers. Figure 34 shows a flow diagram to illustrate this approach.

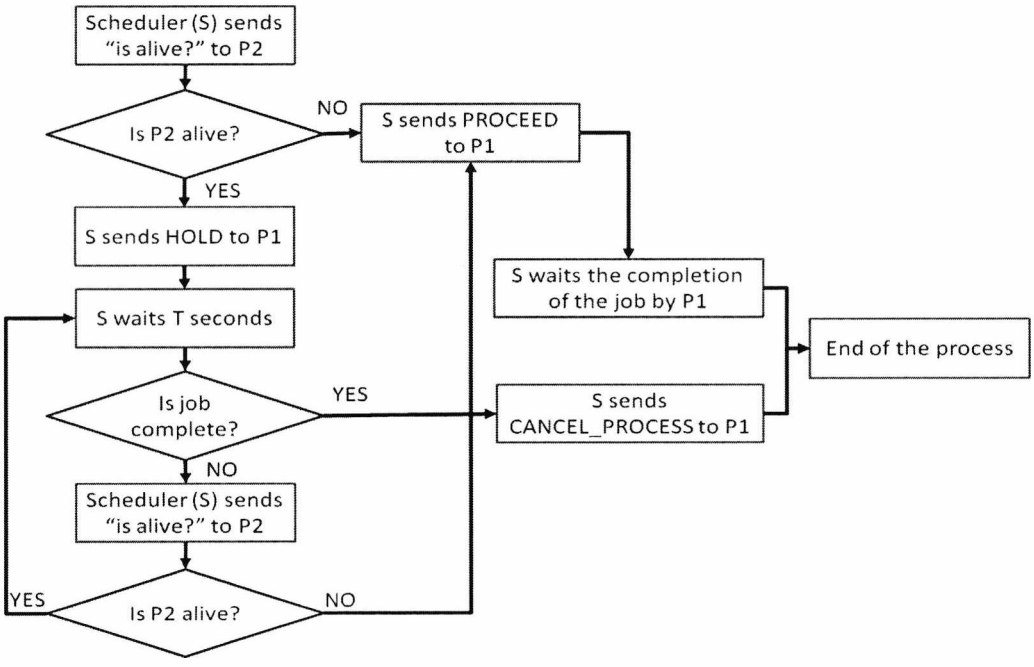

Figure 34 - A solution to avoid job replication using "is alive?" messages.

3. The scheduler may send the HOLD message to P2, which may respond to the scheduler with a HOLD_SUCCESS message, which makes the scheduler send a PROCEED message to P1; or P2 may respond with a JOB_COMPLETION_SCHEDULER message, which indicates that the

job has been executed and the results are being sent to the consumer. If the JOB_COMPLETION_SCHEDULER message is received, the scheduler then sends a CANCEL_PROCESS message to P1, allowing it to process other jobs. This alternative would also avoid the need for concurrency control for writing the results, although it may waste P2's processing power if it is not allowed to process other jobs during the "hold" period; in any case, P1 may not be reliable and may crash again or the results may take too long to be sent to the consumer/repository due to the size of the data, network congestion or to unavailability of the consumer/repository. If the scheduler gets a connection timeout while sending the HOLD message to P2, the scheduler sends the PROCEED message to P1. However, it may have happened that P2 has already finished the submission of the result, but was not able to send the final message to the scheduler. In this case, the consumer or the shared repository should be able to tell P1 that it does not need to resubmit the result, but only to send the final message to the scheduler. Figure 35 shows a flow diagram to illustrate this approach.

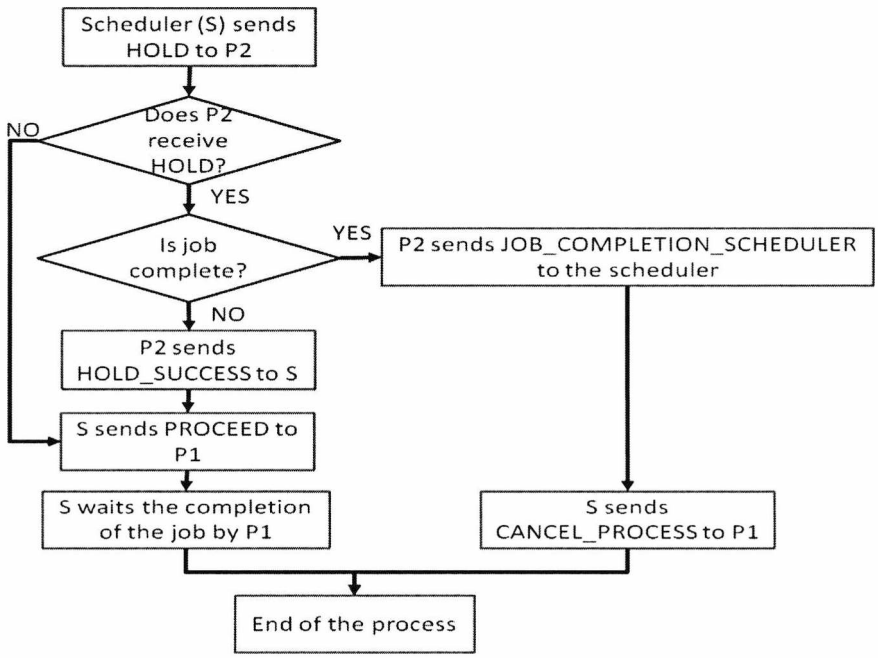

Figure 35 - A solution to avoid job replication that gives preference to the first provider that has completed the job.

4. Another option would be to allow both providers to run the job and to make use of concurrency control strategies. If the consumer specifies another location (e.g., a remote database, a tuple space, data grid) for the data to be stored, we assume that the chosen repository must handle the concurrency control. If the consumer is responsible for receiving the results, we can use an Optimistic Locking strategy [89] for concurrency control, where the systems do not try to avoid collision, but identify them and take further action. In this case, once the consumer identifies that 2 providers are sending the results, it might simply receive from both and ignore one; or it might ask the last one to send a message to wait, giving the "writing lock" to the first provider. In cases where a lock is used, the consumer must use some mechanism to identify any crash of the device holding the lock (e.g. use a timeout if the lock holder does not send any data for a certain period). If a crash of the lock holder is identified, the consumer then gives the writing lock to the other provider. The use of Optimistic Locking is recommended for systems with low collision rates and is scalable when compared to the Pessimistic Locking strategy. With the latter strategy, the write lock would always be given to the first provider, and would block any other provider to write the result, and the consumer itself would not be able to read it until the write lock has been released. The Pessimistic Locking strategy is easy to implement, but does not scale when multiple entities are trying to access the data at the same time. Since we only expect the consumer itself to read the data once it has been written, Pessimistic Locking emerges as a good alternative for our system.

Due to time constraints, it was not possible to evaluate all these options in order to find out which one of them produces the best performance results; hence, we have opted for using the simplest solution (1) in the simulations presented in chapter 5. The evaluation of these alternatives and the proposition of others are left as future work.

4.5.7 Network Partition

Our proposed system does not present a specific mechanism to identify that the network has been partitioned. In fact, if the partition lasts only for a short period, the system may not even identify the failure and continue to work as if nothing has happened.

Long lasting partitions are more likely to require further activity from the components in the grid, since they allow more time for those devices in the partitions not containing the active scheduler to see timeouts and then detect the absence of a scheduler.

When this happens, those devices will attempt to elect a new scheduler. If there are no candidates for the role, the devices will keep periodically trying the election, which will stop it under one of the following circumstances:

- a candidate appears in the partition and a new scheduler is elected;
- the partitions merge and an active scheduler from the other partition replies to the attempt of election;
- a JOB_DESCRIPTION message (broadcast by consumers immediately after receiving the global ID from a scheduler) arrives, so that the devices will try to register themselves with the scheduler that issued the global ID for the job described in that message (this may also mean that the partitions have merged).

When the separated devices manage to elect a new scheduler, there will be a scheduler active in each partition, and when the partitions merge again, the participants should be able to identify the presence of the two schedulers, so that further action can be taken in order to keep only one of the schedulers active.

Assume that after the partition, two grid systems G1 and G2 became active. After the merge, devices in G1 can receive any JOB_DESCRIPTION message from G2, and vice-versa. Consumers in one active grid will discard these broadcast messages. However, providers should always check whether the scheduler that issued the described job is the same one that they know.

If the scheduler is different, it means that there is a high possibility of two schedulers being active (as mentioned above, JOB_DESCRIPTION messages are sent immediately after consumers communicate with their scheduler). In this case,

providers that identified the problem should send a MERGE message to the schedulers, informing them the presence of each other.

The reasons why only providers identify the merge in active grids are:

- We want to avoid any extra processing for consumers and they will not become inconsistent with the presence of another scheduler;
- If providers are allowed to continue working with the presence of more than one scheduler, they will add jobs to their local data structures that will not be recognized by their own scheduler, and then be removed after they receive the SCHEDULE_REQUEST_RESPONSE message from the scheduler. This will only add extra processing for the providers with the matchmaking and extra usage of the network with the transfer of job descriptions that will be discarded.

The pseudo-code in Figure 36 extends the one presented in the Figure 18 by adding the check whether the scheduler that registered the job is the same as the one known by the provider processing the job description. The lines 20 to 23 implements the submission of the MERGE messages to both active schedulers.

```
1. handleJobDescription(job) {
2.     if (job.scheduler IS currentScheduler) {
3.         local_status = coordinator.getStatus();
4.         processable = matchmake(local_status,
5.             job.DESCRIPTION);
6.         if (isProcessingJob AND job IS chosenJob) {
7.             if (processable IS FALSE) {
8.                 stopExecutingJob();
9.                 send(CHANGE_STATUS, scheduler, job.GLOBAL_ID);
10.            }
11.        } else {
12.            if (processable IS TRUE) {
13.                classifyAsProcesable(job);
14.            } else {
15.                classifyAsNotProcesable(job);
16.            }
17.            jobs.add(job);
18.            requestSchedule();
19.        }
20.    } else {
21.        send(MERGE, job.scheduler, currentScheduler);
22.        send(MERGE, currentScheduler, job.scheduler);
23.    }
24. }
```

Figure 36 – Updated pseudo-code for the providers handling the job description.

Once the schedulers receive the MERGE message, they compute their utility values (using the formulae presented in section 3.4.2) and send them to each other (using the MERGE_UTILITY message). They also send the size of the local data structure, which indicates the number of jobs known by the scheduler and yet to be completed, representing an additional factor for the utility value (f_5).

Each scheduler compares its own f_5 value and the one with smaller value increases the utility value received from the other scheduler by 1. This procedure corresponds to applying the category (1) of normalization with max_5 equals to biggest f_5, and min_5 equals to the smallest f_5. The idea is that the scheduler with the highest number of jobs is preferred, since it would avoid more usage of the network with the transfer of the data.

The scheduler with the lowest utility value (considering f_5) stops its activity as a scheduler and sends a SYNC message with its own data structure to the winning scheduler. After receiving the SYNC message, the winner scheduler updates its own data structure, and then broadcasts a MERGE_COMPLETE message that carries the information about the winning scheduler, so that the components can update their own information about the scheduler.

The *handleMerge* procedure in the pseudo-code in Figure 37 implements the behaviour of the schedulers when they receive the MERGE message. Here we also add a random number from 0 to 1 to be used to decide which scheduler should be the winning one, in the case where both computed utilities have the same value, as we show in the *handleMergeUtility* procedure.

In the lines 24 and 30 of Figure 37, the scheduler with higher utility starts a timer to measure the time it has to wait for the SYNC message from the losing scheduler. If after T seconds the SYNC message does not arrive, the winning scheduler assumes that the other scheduler has crashed and broadcasts the MERGE_FAILED message (see the *handleSyncTimeout* procedure in Figure 37). If, for some reason, the winning scheduler fails before the losing one can complete the submission of the SYNC message, the losing scheduler then broadcasts the MERGE_FAILED message (see the *handleConnectionTimeout* procedure in Figure 37).

When a device receives the MERGE_FAILED message, the coordinator assume the sender as the new scheduler, as shown in the pseudo-code in Figure 38 (*handleMergeFailes* procedure). The providers that had a different scheduler before the merge process will have to resubmit their pending jobs and completed jobs

queues (*sendMergeRecoverQueue* procedure). Consumers will not need to take further action, so that we can avoid further processing on them.

Since the broadcast channel is not reliable, some components may not receive the MERGE_COMPLETE message and attempt to communicate with the loser scheduler (now inactive). The inactive scheduler then replies to the sender with a MERGE_COMPLETE message using a reliable channel.

4.6 Summary

In this chapter, we have described our proposal for a scheduling system where the scheduler can run on limited devices. The proposal includes de description of the registering of devices, the discovery of resources, and the submission, scheduling and execution of jobs.

We have also explained how our scheduling system deals with failures, including the proposition of an election mechanism to determine a new scheduler when the current one fails.

The fault tolerance mechanism that aims to avoid job replication still requires further investigation to define the most appropriate solution. In this chapter, we have listed four possible solutions for this problem and have decided to use the simplest one in our simulations, but we cannot guarantee that this solution provides the best performance results, and further performance tests are required.

In Table 4, we present a brief summary of the messages in the protocol. The messages are listed following the order of appearance in this chapter.

In the Appendix C at the end of this thesis, we present all the pseudo-codes used in this chapter organized by the designated components.

```
1. handleMerge(other_scheduler) {
2.     mergeUtility = computeMergeUtility();
3.     size = pendingJobs.size + completedJobs.size;
4.     mergeRandom = RANDOM(0, 1);
5.     send(MERGE_UTILITY, other_scheduler, mergeUtility,
6.          size, mergeRandom);
7. }
8.
9. handleMergeUtility(message, received_utility, received_size,
10.           received_random) {
11.       size = pendingJobs.size + completedJobs.size;
12.       if (size > received_size) {
13.           mergeUtility = mergeUtility + 1;
14.       } else if (received_size < size) {
15.           received_utility = received_utility + 1;
16.       }
17.       if (mergeUtility IS received_utility) {
18.           if (mergeRandom IS received_random) {
19.               send(MERGE, message.sender, THIS);
20.           } else if (mergeRandom < received_random) {
21.               send(SYNC, pendingJobs, completedJobs);
22.               stopLocalScheduler();
23.           } else {
24.               startSyncTimer();
25.           }
26.       } else if (mergeUtility < received_utility) {
27.           send(SYNC, pendingJobs, completedJobs);
28.           stopLocalScheduler();
29.       } else {
30.           startSyncTimer();
31.       }
32.   }
33.
34.   handleSyncMessage(pendingJobs, completedJobs) {
35.       stopSyncTimer();
36.       updatePendingJobs(pendingJobs);
37.       updateCompletedJobs(completedJobs);
38.       broadcast(MERGE_COMPLETE);
39.   }
40.
41.   handleSyncTimeout() {
42.       broadcast(MERGE_FAILED);
43.   }
44.
45.   handleConnectionTimeout(message) {
46.       if (message IS SYNC) {
47.           broadcast(MERGE_FAILED);
48.       }
49.   }
```

Figure 37 - Pseudo-code for the schedulers handling the merge process.

```
1. handleMergeComplete(message) {
2.     currentScheduler = message.sender;
3. }
4.
5. handleMergeFailed(message) {
6.     currentScheduler = message.sender;
7.     provider.sendMergeRecoverQueue();
8. }
9.
10.   sendMergeRecoverQueue() {
11.       send(MERGE_RECOVER_QUEUE, currentScheduler,
12.           jobs, completedJobs);
13.   }
```

Figure 38 - Pseudo-code for the merge completion and failure messages.

Table 4 - The summary of the messages used in the proposed protocol.

	Message	Description
1	REGISTRATION_REQUEST	Message that is broadcast by the components to find the location of the scheduler and inform their ID to it.
2	REGISTRATION_RESPONSE	Message sent by the scheduler to inform the components that the registering process is complete.
3	EXISTING_ID	Message sent by the scheduler to inform the registering components that it should generate a new ID and try again.
4	JOB_SUBMISSION_REQUEST	Message sent by the consumers to submit a job to the scheduler.
5	JOB_SUBMISSION_RESPONSE	Message sent by the scheduler to the consumer carrying the job's global ID.
6	JOB_DESCRIPTION	Message broadcast by consumers to distribute the job description to the providers.
7	SCHEDULE_REQUEST	Message sent by the providers to the scheduler to request the scheduling of a job.
8	SCHEDULE_RESPONSE	Message sent by the scheduler to the

		providers with the result of the scheduling process.
9	JOB_COMPLETION_CONSUMER	Message sent by the providers to the consumers to inform the completion of a job.
10	JOB_COMPLETION_SCHEDULER	Message sent by the providers to the scheduler to inform the completion of a job.
11	CANCEL_JOB	Message that consumers send to the scheduler to cancel a job.
12	SCHEDULER_FAILURE	Message broadcast by the components to indicate that the scheduler has failed and that the election of a new scheduler should begin. This message also carries the utility value of the device.
13	RESTART_ELECTION	Message broadcast by the components to restart the election after their timers reach the timeout and a new scheduler has not been elected.
14	HIGHER_UTILITY	Message broadcast by a scheduler-capable component with a higher utility than the one received from another component.
15	SCHEDULER_ELECTED	Message broadcast by the elected scheduler to indicate the end of the election process.
16	RECOVER_ID	Message sent by the consumers to the scheduler to inform the highest global job ID that they know.
17	RECOVER_QUEUE	Message sent by the providers to the scheduler to inform the jobs that they know.
18	RECOVERY_COMPLETE	Message broadcast by the scheduler to

		indicate the end of the recovery process.
19	CHANGE_STATUS	Message sent by the providers to the scheduler when they are not able to complete a job, so that the scheduler can set the status of job to "available".
20	CONSUMER_UNAVAILABLE	Message sent by the providers to the scheduler to indicate that a particular consumer is not available.
21	JOB_PENDING	Message sent by the providers to the scheduler to indicate that a particular job has been completed, but with the results pending submission to the consumer.
22	PROCEED	Message sent by the scheduler to the providers to allow them to continue with the execution of a job in case of the same job being allocated to more than one provider.
23	HOLD	Message sent by the scheduler to providers to request the latter to suspend the processing of a particular job.
24	CANCEL_PROCESS	Message sent by the scheduler to the providers to request the latter to stop the processing of a particular job.
25	HOLD_SUCCESS	Message sent by the providers to the scheduler to indicate that the suspension of the job processing was successful.
26	MERGE	Message sent by the providers to co-existing schedulers, so that they can start the merging process.

27	MERGE_UTILITY	Message that co-existing schedulers send to each other to inform their own utility values in order to define the winning scheduler.
28	SYNC	Message used by the losing scheduler to send its own data structure to the winning one.
29	MERGE_COMPLETE	Message broadcast by the winning scheduler to indicate the end of the merging process.
30	MERGE_FAILED	Message broadcast by the scheduler that is not able to communicate with the other one during the merge process.
31	MERGE_RECOVER_QUEUE	Message sent to the scheduler by the providers that used to communicate with the scheduler that crashed during the merge process.

Chapter 5. Evaluation of Scheduling Performance

5 Evaluation of Scheduling Performance

In this chapter, we present how the evaluation of our proposed scheduling mechanism was carried out; the results of this evaluation are presented in the next chapter.

5.1 Methodology

There are several ways of evaluating this kind of system. One of them is by the use of simulations, where we can determine the behaviour of the system and study its behaviour without having to implement it, saving time and effort.

Another common evaluation method is formal verification techniques, such as model checking, which can verify a design statically and automatically without test benches [90]. Compared with simulation, the superiority of model checking is full automation with useful counterexamples as by-product. However, state explosion that constantly occurs in large-scale systems is the main problem in model checking, making it difficult to complete the verification.

Another advantage of simulations is that it is easier to collect statistical data on performance to compare with the data from other systems, which makes it the first choice to evaluate systems in the literature.

Since we are also interested in the performance of the proposed meta-scheduler in spite of its correctness, we also opted for using simulations to compare our system with OLB (as explained in Chapter 6). Most of this chapter is dedicated to describing how our simulations were designed.

We have also used model checking to verify our system for some cases of network partition (see Chapter 7). Formal verification was chosen in this case because we do not compare our solution with others, since other grid systems do not provide any special fault tolerant mechanism for this kind of failure. Another reason was the time constraints, since it would take longer to implement these cases fully in the simulation framework.

Even though the model checking was used for a specific purpose, it also helped us to locate and fix problems with the protocol that could not be checked with the simulations, contributing to the correctness of our system.

5.2 Discrete Event Simulation

For the evaluation of this proposal we used discrete event simulation [91, 92], in which every change in the state of the model (e.g. sending a job, transferring a packet, etc.) corresponds to a time-stamped event placed in a queue to be executed serially and chronologically.

We have studied some simulations tools for grid systems: SimGrid [93], GridSim [94], EDGSim [91] and GangSim [95]. Most of them use the discrete-event concept. Some of them were developed for a specific purpose; GridSim, for example, focus on the simulation of grid economy, where the resource management is done based on concepts of Economics. Some of these tools support the use of real workloads for the simulations and the majority only supports the simulation of centralized job scheduling.

These tools have their own network model, and usually they are based on the Internet. We could not find any support for broadcasting messages in the network model of the tools studied. Since our proposal uses broadcast, we had to develop our own grid simulation tool.

Our tool was built using a discrete event simulation framework internally used in the School of Computing at the University of Kent [96]. The entities (simulation objects) in the simulation interact with each other through model queues as shown in Figure 39.

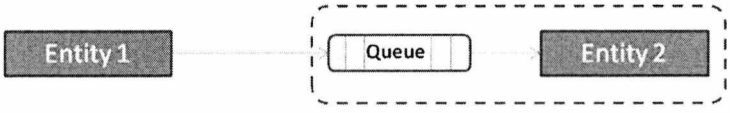

Figure 39 - Interaction between objects in the simulation.

The interaction between Entity1 and Entity 2 is described in terms of work items, which represent the action to be performed, and carry some information such as the start time and the size of the task to be performed.

There is also another queue, which is not directly visible to the modeller, and is accessed by the simulation core. This queue contains the events which, in themselves, take no time, but typically represent the completion of some activity (work items), such as a processing step, a communications process, or a time-out. Each event marks a change in the state of the system, and it will typically enable new

activities leading to the later occurrence of further events. The sequence is continued for as long as necessary to collect a view of the average behaviour of the system.

The events queue is updates by the simulation core when the simulation objects request the core to schedule/unscheduled an event. The events are ordered by their due time, which is computed by the simulation objects and provided by them to the core component.

The simulation core also maintains the clock of the simulation by repeatedly setting the current time to the due time of the next event to happen, when removing it from the events queue.

The simulation model also has a component responsible for collecting statistical data, such as average time between jobs, average delay on the network, etc.

All the objects that have been modelled in this thesis, and the relation between them, will be described further in this chapter.

5.3 Benchmarks

In order to simulate real devices, we first ran some benchmarks to collect real cost values for our simulation. The results of the benchmarks are presented in this section.

Processing Power - MFlops

To check the processing power, in terms of floating pointing operations, of some potential devices for the proposed computational Grid, we decided to run a series of benchmark tests with those devices we have available to create our test environment. For this evaluation, we have used the Linpack Benchmark [97], which consists of solving a dense 500x500 system of linear equations with one right hand side, Ax=b, in which the matrix A is generated randomly. This benchmark is very suitable for the purpose of our project, since, as mentioned before, one possible application for home grids involves SVD, which is also a computation over matrices.

The devices included in this benchmark and their respective specifications are:

- Set-top box (STB): a Philips TriMedia MHP Prototype 2000 with processor TM1300 running at 160 MHz, 32 MB SDRAM, 8 MB Flash Memory;
- Pocket PC: HP iPAQ with Intel® PXA255 processor, 64 MB of RAM memory and 32 MB of ROM flash memory, running Windows CE 4.2 and

the Mysaifu Java Virtual Machine (JVM) [98], a free JVM for PDAs and Pocket PCs that aims to conform with the Java 2 Standard Edition (J2SE);

- Laptop HP: processor AMD Athlon XP 64-bit, 1.5 GHz of RAM memory, running Windows XP Home Edition and Java 2 Standard Edition (J2SE) 1.6;
- Laptop Dell: with an Intel Centrino Duo processor, 1 GB of RAM memory, running Windows XP Professional and Java 2 Standard Edition (J2SE) 1.6.
- Smart phone Nokia N95: processor ARM11-based TI OMAP2420 processor running at 330 MHz, 64 MB of RAM memory, running Symbian OS v9.2 S60 3rd Edition Feature Pack 1 and supporting Java Micro Edition (J2ME) - MIDP 2.0 profile.
- PlayStation 3 (PS3): game console equipped with a PowerPC-base Core processor, running at 3.2 GHz, having 512KB L2 cache, 256MB XDR Main RAM at 3.2GHz and 256MB GDDR3 VRAM at 700MHz; it also contains a 60GB hard disk and can communicate via Bluetooth, USB2.0, IEEE 802.11b/g and Ethernet. Linux 2.6.23-rc6 is the operating system installed in this device and SableVM version 1.13 [99] is the Java Virtual Machine.

In our experiment, we have modified the Linpack Benchmark source code to make it run 100 (one hundred) times, so we could collect all the execution times and compute the average time for each device and avoid randomness in the results. The results are presented as a bar chart in Figure 40.

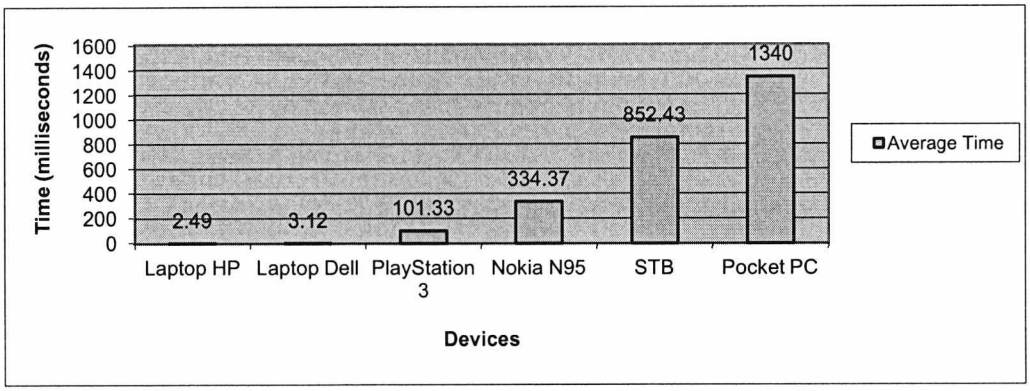

Figure 40 - Average time of Linpack Benchmark

From these results we can also conclude that the average time for limited devices like the Pocket PC and the STB are much higher than the time for more powerful

devices like the laptops. It is important to remember that when the benchmark was executed in the STB, no media content was being received/decoded, and loading from this could increase its average time. These results also give us an idea of which devices are capable of giving some contribution of processing power to the grid.

Processing Power - Mips

We also need to run a benchmark that gives us some values for common operations that are not based on float point processing. For this purpose we used the Dhrystone Benchmark, which is well described in [100].

The java implementation of the Dhrystone benchmark provided in [101] was used in this thesis. The results are expressed in VAX Mips, which represents the number of Mips (Million Instructions per Second) compared to a VAX 11/750 machine. These results are presented in Figure 41.

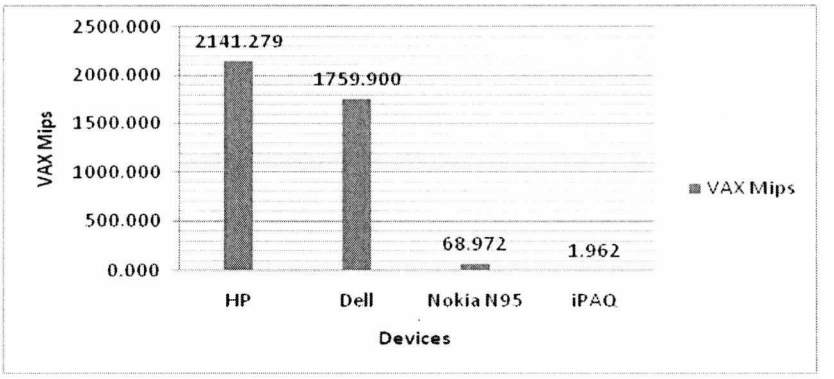

Figure 41 - Dhrystone benchmark results.

5.4 General Simulation Model

In order to simulate the grid with different scheduling mechanisms, we have adopted the abstract model for communicating nodes presented in Figure 42.

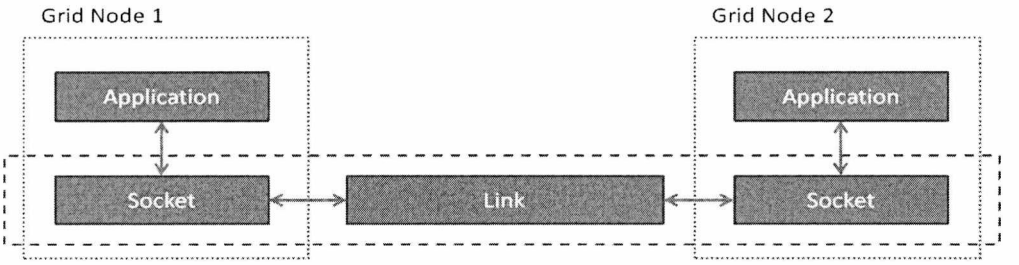

Figure 42 - Abstract Simulation Model.

The idea is to characterize the grid nodes (the devices that are participating in the grid: consumers, providers and scheduler) by the modules:

- **Application**: for consumers, this module corresponds to the generation of jobs that are sent to the grid to be executed; in the case of providers, this module acts like an component of the grid, coordinating the requests for resources and jobs execution within the device hosting the provider; for the scheduler, it represents the scheduler itself, allocating jobs to providers.
- **Socket**: this module is an abstraction for the implementation of a communication protocol. It is responsible for receiving a message from the application and transmitting it through the network using a pre-defined protocol (e.g. TCP [102] and UDP [103]).
- The Link component represents an abstraction of the communication between grid nodes. In this thesis, this module is realized as presented in Figure 43.

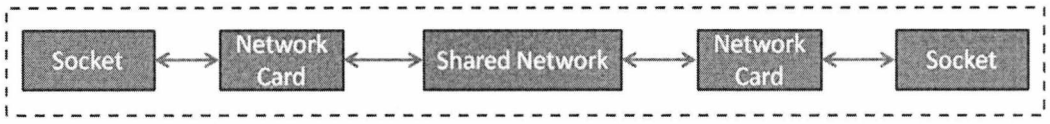

Figure 43 - Realization of the Link component.

- **Network Card**: this is the component that is responsible for sending and receiving packets through the simulated network. For this thesis, network cards are characterized by the bandwidth for the communication. With this component as part of a grid node we are able to simulate devices with different network capabilities.
- **Shared Network**: this module, in this thesis, is assumed to be an active wireless Router.

Grid nodes are also characterized by a **Processor** module, which represents the processor of a device in terms of its computational power. In this thesis, we are describing this computational power according to the following two parameters values: MFLOPS and MIPS. The first value indicates the number of float point operations the processor can execute per second; the second represents the number of general instructions that can be executed per second by the processor. These two values are used to calculate how long each computation task in the simulation should

last. This module works with the **Local Scheduler** module, which represents the operating system scheduler and controls the sequence of execution of the tasks in the grid node's processor.

This modularization of the simulation model allows us to change the implementation of a certain module (the network card and/or socket, for example) without affecting the other modules.

5.4.1 Consumer

Each consumer generates groups of jobs that are sent to the grid. Each group of jobs has a maximum number of jobs (*GROUP_SIZE*) that can be sent to the grid sequentially, without having to wait for the previous one to be completed. After sending *GROUP_SIZE* jobs, the consumer application has to wait until all of them have been executed, before it starts sending another group. The interval between sending two consecutive jobs in the same group is computed by using a Uniform Distribution that gives Real values between 0 (zero) and α seconds of simulated time. In most of the results presented in Chapter 5, we have used $\alpha = 0.5$, which gives an average of 0.25 second of simulated time between job submissions.

After executing all the jobs of a certain group, the consumer application waits for an interval defined by an Exponential Distribution and then starts submitting another group of jobs. The parameter λ for this distribution determines the load of the simulation. The average waiting time W is given by:

$$W = \frac{1}{\lambda}.$$

So, for example, if λ is set to 0.01, it will give an average interval of 100 seconds of simulated time between groups submission; lower values for λ reduce the average waiting time, increasing the number of groups submitted and, consequently, the load of the simulation.

5.4.2 Jobs and Work Items

In our simulation model, jobs are described by their computational size (*Job_cs*) expressed as a number of instructions, the input data and executable program size, and the resource requirement (minimum CPU speed, storage and network bandwidth, the type of Operation System, and the power source – indicating if devices running on batteries are allowed).

The job also specifies the proportion of common operations (*mips_factor*) and the proportion of floating point operations (*mflops_factor*). When the Processor of a grid node receives a job to be processed, it computes the duration of processing the job with both Mips (*Pmips*) and MFlops (*Pmflops*) components using the following formulas:

$$\text{I.} \quad duration\_mips = \frac{Job\_cs}{Pmips} \times mips\_factor$$

$$\text{II.} \quad duration\_mflops = \frac{Job\_cs}{Pmflops} \times mflops\_factor$$

In the formulas I and II, *duration_mips* represents how long the job would take to be processed if it uses only the Mips component of the Processor; while the *duration_mflops* indicates the duration of a job being processed only by the MFlops component of the Processor.

To compute the total time of processing a job (*duration*), the formula III is used:

$$\text{III.} \quad duration = duration\_mips + duration\_mflops$$

The work item is a component that carries temporal and statistical values related to all the steps of processing each job in the simulation model (e.g. the initial and final time of execution).

5.4.3 Size of messages

To estimate the size of data being transmitted between grid nodes, we use the number of bits that is most commonly used to represent the type of that data (e.g. integers are represented by 32 bits, while characters are expressed by 8 bits). The exception is the size of the input data used by the jobs, which is generated using a power law (Pareto) distribution, which is reported to be appropriate to model file sizes [104].

For example, in our simulation, a job is represented by its description, the job itself and the input data. The description consists of the GUID of the sender (which corresponds to 128 bits), of an ID of the job (represented by an integer – 32 bits) and the following values referring to the requirements for the job:

- CPU speed (Number of instructions): represented by a double value (64 bits); in our model, we specify this information in terms of Mips (Million

Instructions per Second) and MFlops (Million Floating Point Operations per Second);

- Network Bandwidth (KBps): also characterized by a double value (64 bits);
- Battery: specified by a Boolean value (in 8 bits) that indicates if the grid node can be running on battery or not;
- Disk space (rotations per second): described by a double value (64 bits);
- Operating System (OS): represented by a byte value (8 bits); the value for each OS is pre-defined.

Considering only the values, we assume that the size of the description of jobs is 368 bits, which corresponds to 46 bytes. However, a piece of information identifying the values is also needed, for example, a tag to identify that a value FALSE corresponds to the requirement related to the battery usage. For this reason, we arbitrarily added 50 bytes to the size of the job description, resulting in a total of 96 bytes.

Values for the size of the executable binary and for the input data can also be defined in our simulations, and they are set according to the experiment being performed, but are usually a lot more than 96 bytes.

5.4.4 Network Simulation

In order to send and receive messages, the applications in the simulation make use of sockets. As mentioned before, the sockets in this simulation are the implementation of communication protocols. For this thesis we use a simplified implementation of the User Datagram Protocol (UDP) [103] to broadcast messages (e.g. the job description sent by consumers), and a simplified version of the Transmission Control Protocol (TCP) [102] for any other type of communication.

In this thesis, sockets are implemented as pairs of socket components: one for sending packets (Sender Sockets) and another to receive packets (Receiver Sockets). Sender Sockets receive the message from the application, break it into small packets and forward these packets to the Network Card Output. The Receiver Sockets receive packets from the Network Card Input and, when all the packets of a certain message are received, the receiver socket then forwards the message to the

corresponding application. Figure 44 illustrates the connections between these communication components.

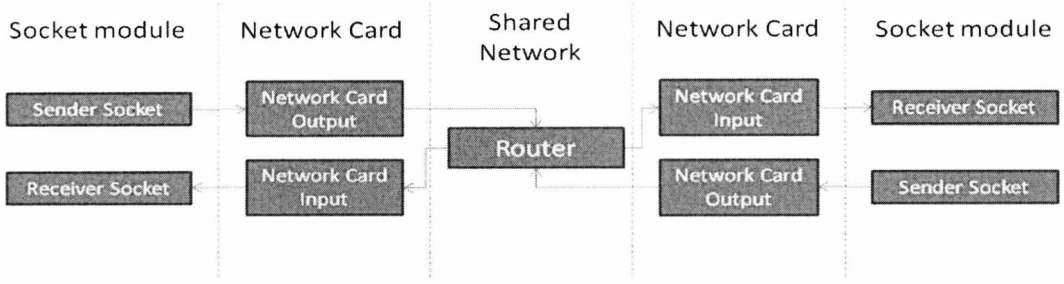

Figure 44 - Connections between communication components

TCP Sockets

Here we use a simplified version of TCP that suits the purpose of our research. Our implementation of TCP uses a simple *positive acknowledgment with retransmission* (PAR) technique where for each packet sent, a positive acknowledgement must be received before the next packet can be sent (using retransmission when needed). Real TCP uses sliding window technique, which allows many packets to be sent at once and resubmits those that were not acknowledged, improving the communication performance.

When grid node A needs to send a message using TCP to grid node B, A's Sender Socket (SSA) breaks the message into packets and starts sending them. After submitting a packet to the network, SSA must wait for an ACK packet. ACK packets are sent by the destination's Receiver Socket, which places them into the Network Card Output queue. So, when B's Receiver Socket (RSB) receives a packet from SSA, it places an ACK packet in B's Network Card Output. This ACK packet will be received by A's Receiver Socket (RSA), which will inform SSA that a new packet can be sent to RSB. This process continues until RSB receives the last packet of the message and sends the last ACK packet. When this happens, RSB recovers the message from all the received packets and forwards this message to the corresponding application. This process is illustrated in Figure 45.

During this communication, an ACK packet may be lost, which would make the socket wait forever. For this reason, we also implemented a retransmission mechanism based on [105], which specifies the timer for retransmission of a packet.

The retransmission mechanism works as follows: for each packet submitted, the Sender Socket schedules a retransmission event to happen after RT (Retransmission Time) seconds. If the ACK is received before RT seconds, the scheduled event is cancelled. If RT seconds elapse before the ACK is received, the packet is then retransmitted, and a new retransmission event is scheduled. But, instead of using RT as the deadline for the next retransmission, RT1 is used, where RT1 = 2 x RT (*back off the timer*). If the ACK is not received after RT1 seconds, another retransmission event is scheduled for (RT2 = 2 x RT1) seconds. Retransmission events will keep being scheduled until an ACK is received or until RTN >= 60 seconds, where RTN = 2 x RT(N-1). If no ACK packet is received, the Sender Socket sends a Timeout message to the application, which will have to act accordingly.

RT is computed based on the average Round-Trip Time (RTT), which is the total time from the submission of a packet until its ACK is received. The average RTT is updated after an ACK message is received based on Karn's Algorithm [106], where the RTT for packets that have been retransmitted are not taken into account.

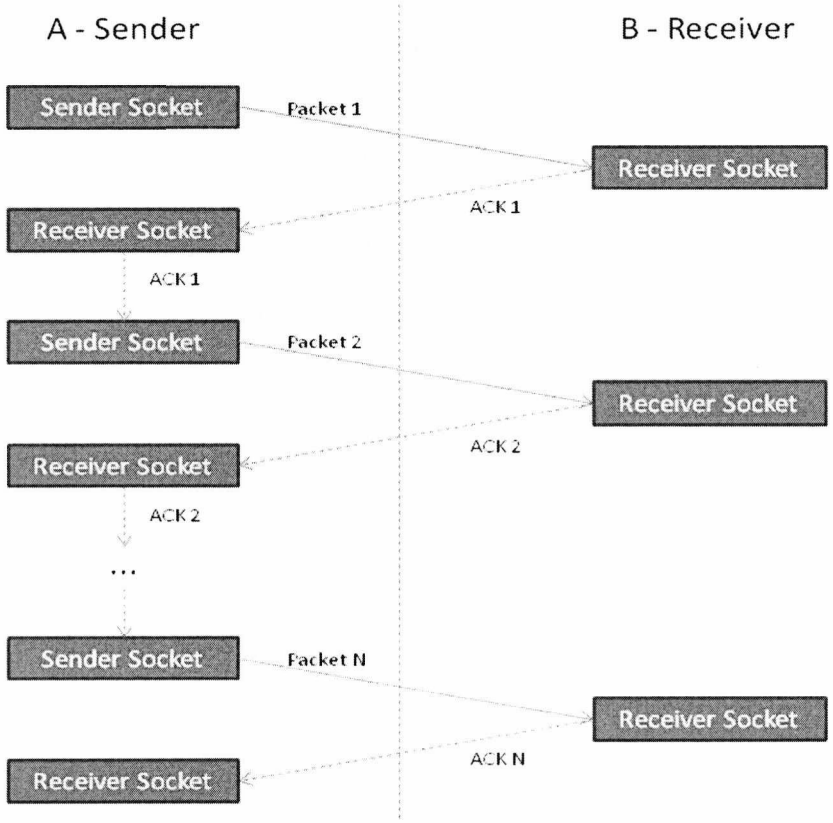

Figure 45 - Simulated TCP communication

UDP Sockets

UDP Sender Sockets are used when the application needs to broadcast messages. This kind of socket still breaks the message into small packets, but they do not need to wait for an ACK packet in order to send the next one, in contrast to what happens with TCP Sender Sockets.

When the Receiver Socket gets a UDP packet, it does not send anything back to the sender. However, when the last packet of a message is received, the Receiver recovers the message and forwards it to the corresponding application.

Receiver Socket

In our simulation model, we consider that an application has one Sender Socket for each grid node it is in communication with. However every grid node application must have only one active Receiver Socket. This type of socket is capable of handling packets from both TCP and UDP Sender Sockets.

Network Router

All the Sender Sockets of a grid node application send their packets to the Network Card Output queue. The network card then sends the first packet on the queue to the shared network (represented by a wireless Router in this thesis). If the Router is busy processing a packet from another grid node, the newly sent packet is put in a waiting queue to be sent when the Router finishes the previous processing.

The packets received by the Router are then sent to the Network Card Input of the grid node the packet was sent to.

When the packet corresponds to a broadcast message, the Router randomly decides which grid node on the list of targets will be the first to receive the packet. After the first one is decided, the Router continues sending the packet to the rest of grid nodes following the order in the list (when the packet is sent to the last one in the list, the next one is the first on the list). For example, suppose we have a list of devices A, B and C that are the possible devices to receive the broadcast packet. If the Router randomly chooses B to be the first device to receive the packet, then the next one will be C, followed by A.

The abstract model of the network simulation described above is shown in Figure 46.

The amount of time (*transmission_time*) that a packet takes to be delivered is computed as follows:

IV. $$transmission\_time = (\frac{packet\_size}{sender\_bandwidth}) + (\frac{packet\_size}{receiver\_bandwidth}) + \gamma$$

The values represented by *sender_bandwidth* and *receiver_bandwidth* are the bandwidth (in Kbytes per second - KBps) of the packet's sender and receiver nodes, respectively; while *packet_size* is the size (in KBytes) of each the packet being transmitted. The value γ is a constant in our model to simulate an internal delay for the router to route a message to the receiver; here we assume that this delay should be very short, since it is just a quick task performed in the router, and arbitrarily set it to 1 microsecond.

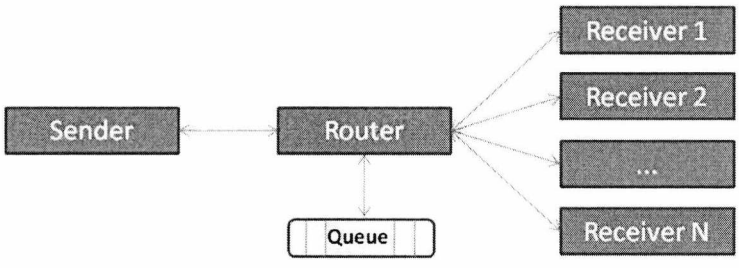

Figure 46 - Abstract model of the network simulation.

For this thesis, we assume that each packet can have the maximum size of 1.496 KBytes that corresponds to 1.456 KBytes of data and 0.040 KBytes of header. These values are based on the TCP protocol.

5.4.5 Processor

The main duty of the processor is to compute the duration for computational activities. This duration is computed using the formula III.

To compute the duration for I/O operations, we use the following factor *f*:

$$f = 9 \times 10^{-5} \text{ seconds/KByte}$$

This value *f* is derived from the I/O benchmark that consisted of writing and reading approximately 80 MB to a file (presented in [107]). The factor *f* is then multiplied by the data size (expressed in KB) to determine the duration of the I/O operation.

In this thesis, I/O operations are performed in the scheduler and the providers in order to store the input data that are used by the jobs they receive.

5.4.6 Number of Instructions

To simplify the process of defining the number of instructions for the scheduling algorithms, we have designed pseudo-codes to help computing the number of instructions for each scheduling algorithm that we want to evaluate. The pseudo-codes were then implemented in C++ and, with the help of the C++ compiler (MingGW [108]), we could generate the Assembly code, so we could count the number of instructions and use it in our simulations. The C++ code can be seen in the Appendix D. Note that, for presentation purposes, we refer to our proposed system as DMS (Distributed Matchmake Scheduling).

Figure 47 shows a pseudo-code for our scheduling model.

```
1. schedule(providerJobsList, n, providerID) {
2.     result = NULL;
3.     i = 0;
4.
5.     while (i < n) {
6.       temp = pendingJobs.get(providerJobsList.next);
7.
8.       if (temp IS NOT NULL) {
9.          if (temp.status IS NOT SCHEDULED) {
10.           if (result IS null) {
11.               result = temp;
12.               result.providerID = providerID;
13.               result.status = SCHEDULED;
14.           }
15.         }
16.      } else {
17.         providerJobsList.remove();
18.      }
19.      i = i + 1;
20.   }
21.   return result;
22. }
```

Figure 47- Pseudo-code for our proposed scheduler.

To demonstrate the expansion of the pseudo-code into assembly code, Figure 48 shows the C++ code that corresponds to the lines 5 and 6 of Figure 47. In Figure 49,

we show part of the assembly code generated by the C++ compiler for the code in Figure 48.

```
1. while (i < n)
```

Figure 48 - Sample of C++ code.

```
1.  movl  -28(%ebp), %eax
2.  cmpl  16(%ebp), %eax
3.  jge   L1
4.  leal  -28(%ebp), %eax
5.  movl  %eax, 4(%esp)
6.  movl  8(%ebp), %eax
7.  movl  %eax, (%esp)
8.  movl  $-1, -152(%ebp)
```

Figure 49 - Sample of Assembly code generated from C++ code.

After analyzing the complete assembly code generated by the compiler, we assumed that the 'schedule' algorithm for the DMS model corresponds to the number of instructions expressed by the formula:

$$V. \quad number\_of\_instructions\_DMS = a + bn + cn\log(N)$$

In formula V, n is the number of jobs in the list sent by providers; and N is the number of jobs in the hash map where the scheduler stores all the jobs that arrive to be scheduled; the constant values a, b and c are derived from the assembly code, where b corresponds to the number of instructions for the loop expressed on the lines 5 to 20 of Figure 47, c is the number of instructions for the implementation of getting a value in the hash map that contains the jobs sent to the scheduler, and a is the number of instructions that are executed outside the loop.

The values that we adopted for the constants a, b and c are presented in Table 5.

Table 5 - Constant values for the instructions of DMS scheduling algorithm.

Constant	Value
A	97
B	250
C	60

For the OLB (Optimistic Load Balance) scheduler, we have adopted the algorithm showed in Figure 50 (pseudo-code).

```
1. schedule(job, m) {
2.     result = null;
3.     temp = new ArrayList(m);
4.     i = 0;
5.
6.     while (i < M) {
7.        if (readyServers[i] == true) {
8.           if (match(job, providers[i])) {
9.               temp.add(providers[i]);
10.          }
11.       }
12.       i = i + 1;
13.    }
14.
15.    n = temp.size;
16.
17.    if (n > 0) {
18.       index = random();
19.       index = index % n;
20.       result = providers[i];
21.       job.status = SCHEDULED;
22.       job.serverID = result.id;
23.    }
```

Figure 50 - Pseudo-code for OLB scheduler.

The same rationale used to define the number of instructions for the DMS algorithm was used for the OLB scheduler, resulting in the following formula for the number of instructions:

$$\text{VI.} \quad number\_of\_instructions\_OLB = a + bM + cm + dm' \tag{VI}$$

In this case M is the number of providers in the grid; m is the number of available providers, i.e. those that do not have any job allocated to them; and m' is the number of available providers that have enough resources to match the job's requirement. The value m is simply managed by the scheduler by decrementing a variable when a job is allocated to a provider and incremented when the provider sends a message informing the scheduler that it has finished the execution of that job and it is available again. The local management of m is an optimization to avoid extra usage of the network in order to find out how many providers are available.

For formula VI, the constant value a means the number of instructions outside the loop represented by the lines 6 to 13 in Figure 50, b is the number of instructions

that should be executed by the scheduler for each provider in the grid, c corresponds the number of instructions executed for each available provider (which includes the instructions for performing the matchmaking process), and d are the instructions for executing line 9 of Figure 50, and are performed for each provider that matches the requirement of the job.

The values that we adopted for the constants a, b, c and d are presented in Table 6.

Table 6 - Constant values for the instructions of OLB scheduling algorithm.

Constant	Value
a	151
b	48
c	181
d	56

The main difference between these algorithms is the matchmaking process, which is performed by the providers in the DMS model, while it is executed by the scheduler in the OLB model. The main part of the matchmaking process in the OLB scheduler is represented by the invocation to the method *match* in line 8 in Figure 50; however, the whole loop from line 6 to line 13 in Figure 50 is part of the process.

Taking a scheduling example where there is one job to be scheduled and only one scheduler available and capable of processing the job, the OLB scheduler would have to process 436 instructions (based on formula VI), while the DMS would require 347 (based on formula V).

When the number of jobs is increased, the size of the DMS's data structure (the scheduler's queue and the list sent by the providers) also increases, requiring more instructions to be executed. However, the size of the data structures reduces more quickly when there are more providers available, which also reduces the number of instructions that have to be executed by the scheduler. All the matchmaking processes in the DMS are done by providers, which we assume that are faster than the device that runs the scheduler, speeding up the process.

Considering only the line 8 in Figure 50, the matchmaking in the OLB would correspond to, approximately, 12% of the instructions executed (56 instructions). When we consider the complete loop from line 6 to 13 in the same figure as part of

the matchmaking process, this percentage increases to, approximately, 65% of the total schedule process; and this percentage becomes bigger when there are more providers available, affecting the performance (see Chapter 6). If we include the costs involved for the scheduler to receive the status information from providers and store it locally, the number of instructions for the matchmaking would increase even more.

To simplify our simulations, we are considering only five values for the matchmaking: CPU speed, network bandwidth, battery usage, storage and operating system. In the C++ code of OLB presented in Appendix D there is an implementation of the method *match*, which corresponds to 53 instructions in the Assembly code generated by the compiler. This number is also used to compute the matchmaking done by the providers in the DMS model.

These estimated values for the number of instructions may be different depending on many factors such as optimizations made by the compiler.

5.5 Scenarios

In order to have a better view of how our proposed model behaves in comparison to OLB, we decided to evaluate three different scenarios as described below:

(i) **Static Scenario**: this scenario consists of simulating an environment where consumers send jobs at a constant rate and all devices in the grid remain connected to the grid until the simulation is completed.

(ii) **Dynamic Scenario**: in this scenario, providers keep connecting and disconnecting from the grid. However, in this scenario, providers do not disconnect while they are executing jobs, in order to avoid faults in the system.

(iii) **Fault Tolerance Scenario**: this scenario aims to evaluate the fault tolerance of the grid system; for this, it extends the Dynamic Scenario by allowing devices to leave the grid at any time (e.g. providers disconnecting while executing a job); the device hosting the scheduler also becomes unavailable in this scenario, which requires the election and recovery of a new scheduler.

In the next chapter, these scenarios are described in more detail, including the implementation and the results of simulations.

The main objective of our scheduler is to maximize the number of jobs completed during a certain period of time. For this reason, one of the variables that are collected from the simulations is the number of jobs completed per second of simulated time.

In order to have a better understanding of the results and to help us in improving our scheduler mechanism, we also collect other variables from the simulation, such as: the mean residence, which corresponds to the total time that a job takes to be completed; the scheduling time, which corresponds to the amount of time that a job takes to be scheduled; and the CPU usage for the scheduler and also for the providers. Other variables are also collected and are introduced when needed to explain the results obtained.

Chapter 6. Simulation Results

6 Simulation Results

This chapter presents the results from the simulations of the different scenarios. These results guided the design decisions leading to the final specification of the fault tolerant scheduling mechanism.

Once we have achieved a stable definition of our protocol, we have continued with the simulations to explore the consequences of our design choices, and to increase the confidence that there are no hidden bugs in the design.

We have decided to evaluate the system under extreme load conditions so that we could verify not only how scalable it is, but also to find external limiting factors. For this reason, we have implemented components for the communication to simulate TCP and an environment using a wireless router instead of using statistical data to simulate the arrival of messages at some components. As shown later in section 6.3.10, the network usage may influence considerably the system's performance, and such test would be much harder to do using statistical data, which may not reflect exactly the load on the communication mean (in this case, the wireless router).

The simulation setups and results in this chapter will demonstrate that the addition of a P2P-based resource discovery process produces acceptable costs compared to the OLB, and a scalable and flexible infrastructure that allows fault-tolerant mechanisms (not provided in the OLB).

6.1 Optimistic Load Balance (OLB)

Considering the requirement for a powerful infrastructure to achieve good performance with job dependency, we assume that the jobs submitted to our grid system are independent (freestanding), since we cannot guarantee such an infrastructure in the home environment. Therefore, the schedulers related to our research are those supporting independent tasks.

The objective of our scheduling solution is to maximize the usage of the available resources for different applications instead of a single application; and we consider a very dynamic and heterogeneous environment. Therefore, we cannot compare our solution with application-centric schedulers and we should avoid static

algorithms, since they require knowledge about all the tasks and resources available in order to produce good results, not adapting well to dynamic changes in the grid.

Considering the statements above, the requirements in section 2.5 (specially D.2) and the taxonomy in Figure 7 in Chapter 3, dynamic schedulers for independent tasks would be the most appropriate for the grid environment assumed in this thesis.

From the dynamic load balancing techniques presented in the Chapter 3, the balance-constrained heuristic requires high network usage, which not only degrades considerably the performance of the system (as shown later in this chapter), but also demands more power from devices running on battery (which goes against the requirements A.1 and A.2 in section 2.5), making this approach not the most suitable for the home environment. Although the cost-constrained heuristic performs better than the previous approach in terms of communication, it requires more processing power for the scheduler in order to compute the costs and make decisions about migrating jobs or not, which is a problem when the scheduler runs on a limited device. The Hybrid heuristic uses static scheduling, which it not very suitable for the scenario studied in this thesis, as explained earlier.

Based on the facts discussed above, we have decided that from the studied scheduling mechanisms, OLB would be the most suitable for the home environment, for its simplicity (with complexity of $O(m)$, where m is the number of resources/providers on the grid), low footprint and adaptability to the dynamic and heterogeneous environment that we envisage. For this reason, we have simulated a version of OLB to use as a benchmark for our proposed scheduling solution.

In order to deal with the heterogeneity of resources, we added a matchmaking process to our OLB scheduler, where a set of QoS requirements are defined for each job, and the scheduler chooses arbitrarily one of the available resources (providers) that match those requirements.

It is common in the literature to present OLB with poor performance results when compared to other heuristics, but this is often an unfair comparison. OLB, which is a dynamic resource-driven (the objective function is to maximize the usage of resources) scheduling heuristic is usually compared against static application-driven (the objective function is to improve the performance of applications) scheduling heuristics. The set-up for the experiments is typically a group of known tasks and the common metric used in those comparisons is the *makespan* [36], which corresponds to the time taken from the scheduling of the first task in the group until

the time when the last task is completed. The *makespan* is most suitable for measuring the performance of application-driven schedulers (where all the tasks are known *a priori*), and it does not reflect the performance of dynamic and resource-driven schedulers, which are best evaluated by the use of throughput or CPU usage as a comparison metric, which are some of the metrics used in this thesis.

One of the differences between OLB and our proposed scheduler is that an OLB scheduler must also perform the matchmaking, while providers perform this process in our solution.

In our implementation of OLB, we assume that whenever a provider comes online in the grid, it should send its current status and ID to the scheduler, so that it can use those pieces of information in the matchmaking process. For the completion of a job, we assume a mechanism similar to our system, where the provider sends the results to the consumer that started the job, and then informs the scheduler that the job has been completed. The difference from our system is that, in OLB, the provider must also send the current status to the scheduler, so it can have the most up-to-date information about the providers, since we cannot guarantee an infrastructure for a GIS (Grid Information Service) that is able to provide updated information about all resources.

6.2 Static Scenario

This scenario is called Static because, during the complete simulated time, none of the devices in the grid becomes unavailable. This kind of situation is simple but not realistic, since devices in the home environment are likely to turn on and off with a certain frequency.

At this stage, we are interested to find out how both schedulers behave in ideal conditions (no disconnections and/or failures) and use this information to make improvements to the design; it is also desirable that our system performs well in this kind of scenario.

First, we make a brief description of the model simulated, including the flow of pieces of information between the participants in the grid.

Note that, for presentation purposes, we refer to our proposed system as DMS (Distributed Matchmake Scheduling).

6.2.1 DMS Version 1

We designed several versions for our system before we achieved the specification presented in the Chapter 4. In the initial version (referred here as DMS1), providers keep only the information about jobs that they are able to process, and always send the complete list of jobs to the scheduler, so a job could be selected by it and the list updated, with the removal of those jobs that have already been executed.

The complete flow of information of DMS1 is shown in Figure 51.

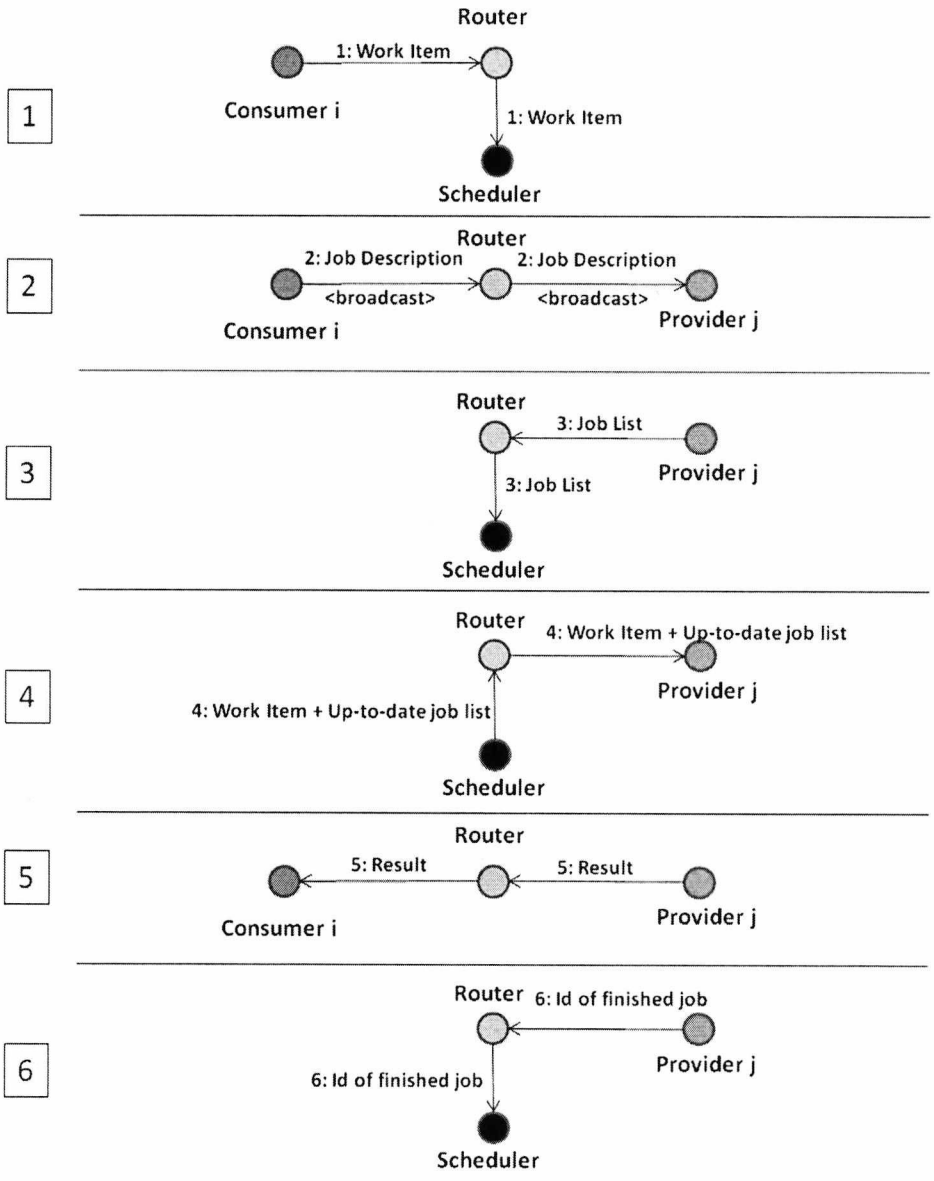

Figure 51 - DMS1's information flow.

Another difference from the final specification in Chapter 4 is that, in DMS1, we did not define a global ID for jobs, since there were no considerations about fault tolerance. It was meant to be a very simple protocol, by avoiding the need for such an ID. This way we could avoid an extra message sent from the scheduler to the consumer.

Since none of the devices disconnects from the grid in the static scenario, we did not need to include the registration process in this simulation.

6.2.2 OLB Information Flow

Here we describe the information flow for the OLB model in the static scenario. As we can see from Figure 52, there are fewer messages exchanged in this model than in DMS1 (see Figure 51).

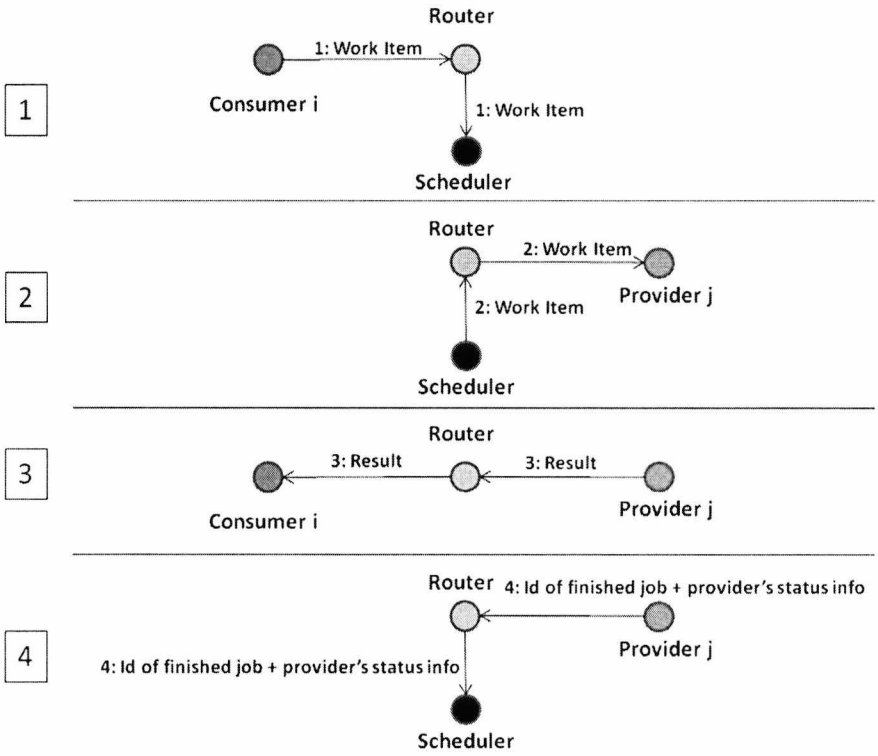

Figure 52 - OLB's information flow.

Since the complete scheduling process (including the matchmaking) is performed by the "Scheduler", no messages are sent from the consumers to the providers.

Similarly to DMS1, there is no global ID for the jobs, and we did not include any registration process for the devices. Note that providers have to send their status

information to the scheduler, since there is not a service in the grid to provide such information, which is used by the scheduler to perform the matchmaking.

6.2.3 Results: DMS1 vs. OLB

The initial aim was to compare the performance of the two systems (DMS1 and OLB) with a high load of job submission, so we could see the point of saturation for each system and how many jobs were completed at the point.

The simulation was run for 100000 seconds of model time, with 10000 seconds of settling time (which is not taken into account for the data collection).

The processing power that we assumed for the providers was the one obtained from the Mips and MFlops benchmarks for the Dell laptop. This device was chosen as a representative high CPU device.

As for the network configuration, we used a bandwidth of 11 Mbps. This value is the same for the communication between all the devices involved in this simulation and it was chosen based on an average bandwidth value from different technologies.

Since we are most interested in the impact of the scheduling mechanism on the number of jobs processed, we have used negligible values (1 Byte) for the input and output data, and a small value (1 KB) for the executable program that will run on the providers. This avoids extra loading on the network, making clearer the influence of the scheduler on the total system's performance.

Increase in the simulation workload can be achieved by augmenting the number of consumers, by changing the size of the group of jobs sent by these consumers (please refer to section 5.4.1) or by changing the parameter λ for the Exponential distribution that determines the amount of time that a consumer should wait before sending the next group of jobs.

For this experiment, we have assumed the existence of 5 providers, and, to boost the load, we have simulated 100 consumers sending groups of 20 jobs to the grid.

The λ parameter (Load Parameter) ranges from 0.01 to 0.09, which means that the average waiting time between groups ranges from 100 to 11 seconds (approximately). Figure 53 shows the comparison between DMS1 and OLB for the number of jobs completed per second of simulated time.

From the results presented in Figure 53 we can see that the performance of the DMS1 starts to downgrade rapidly when the load parameter is greater than 0.03. It also shows that OLB starts to saturate at the load 0.07.

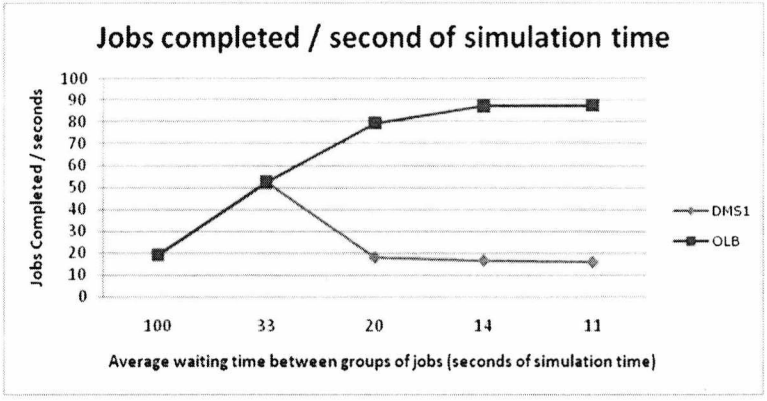

Figure 53 - Jobs completed / second of simulated time: DMS1 vs. OLB.

The downgrade is because a DMS1 provider always sends in full its local job queue to the scheduler (Step 3 on Figure 51); this queue gets bigger when the load exceeds the provider's capacity; the scheduler then requires more CPU and time to process these big queues.

In this trial, the average size of the queue sent by providers has ranged from 2.25 (at load 0.01) to 1750 jobs (at load 0.09), approximately. This increase makes the CPU of the device running the scheduler reach its peak, causing delays in the scheduling process. The percentage of CPU usage for the scheduler is shown in Figure 54.

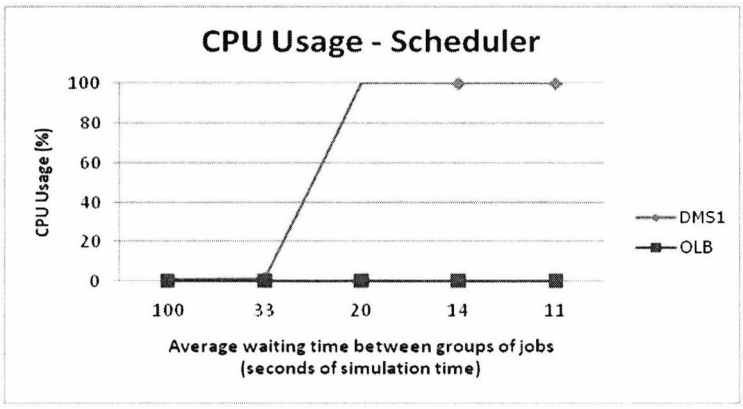

Figure 54 - Scheduler's CPU Usage: DMS1 vs. OLB.

6.2.4 DMS Version 2

In order to improve the performance of the system, we imposed a size limit for the queue sent by providers. Therefore, instead of sending the complete queue of jobs, providers only have to send the first N jobs in their local queues.

With this change, we expect to see a reduction in the CPU usage by the scheduler, a reduced need for bandwidth to send the queues through the network, and an increase in the number of jobs completed.

This new version of DMS is referred in this thesis as DMS2.

6.2.5 Results: DMS2 vs. OLB

To verify the improvements in the performance, we have run a simulation for DMS2, using the same configuration used when comparing DMS1 and OLB.

For this test, we have adopted a limit of 10 jobs in the queue (N = 10), and the graph in Figure 55 shows the number of jobs completed per second of simulated time.

The results show a considerable improvement, with the system completing slightly more jobs than OLB at the highest load of the simulation. In Figure 56, we can see the comparison of scheduler CPU usage between DMS2 and OLB.

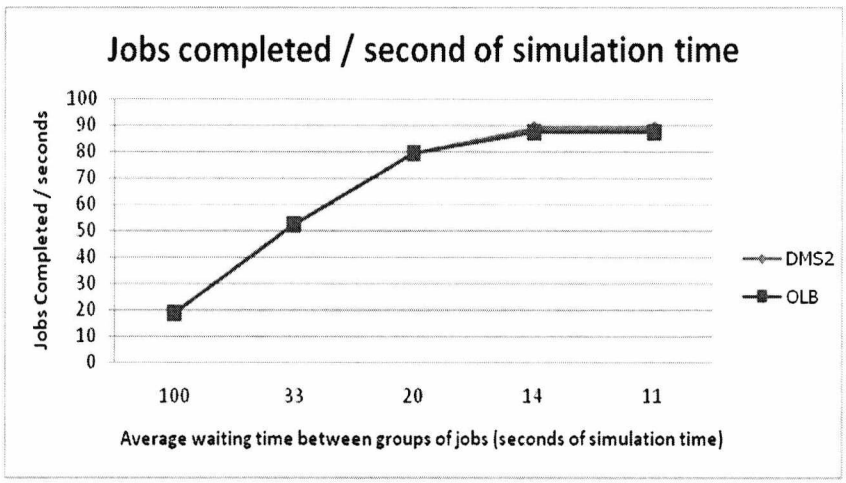

Figure 55 - Jobs completed / second of simulated time (load from 0.01 to 0.09): DMS2 vs. OLB.

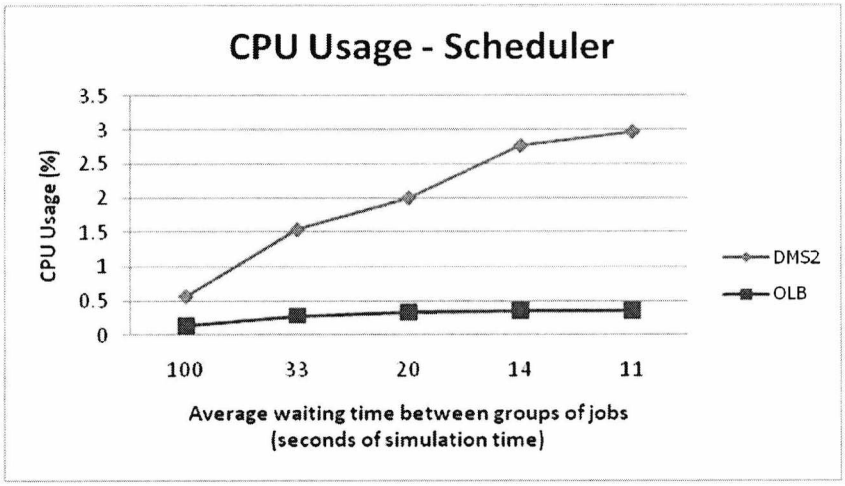

Figure 56 - Scheduler's CPU Usage: DMS2 vs. OLB.

The scheduling time presented in Figure 57 represents the average amount of time that a job take from the moment it is added to the scheduler's queue until it is received by a provider to be processed. Although the DMS2 uses more CPU than the OLB, the DMS2 takes less time to schedule a job. This happens because the OLB depends on the status information that providers send to the scheduler to perform the matchmaking process. The provider's information adds more data to the network and the scheduler is also required to process the information. Delays on the network delay the status information to be delivered and, consequently, holding-up the scheduling decision.

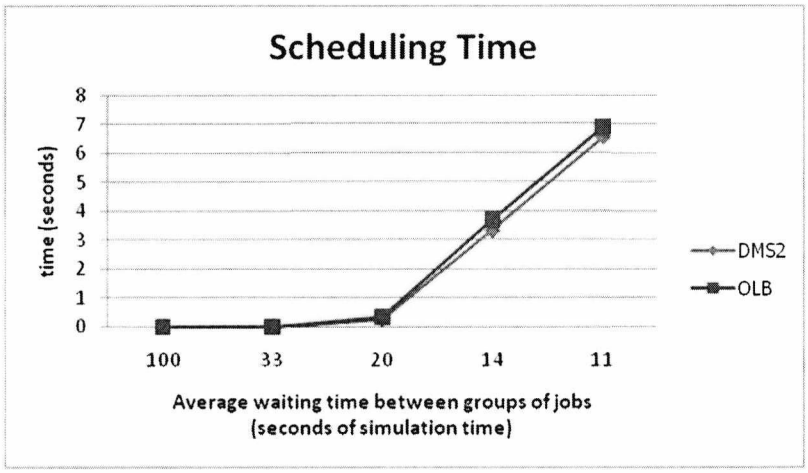

Figure 57 - Scheduling Time: DMS2 vs. OLB.

In this case, considering the average waiting time between groups of jobs of 11 seconds of simulated time, there was an improvement of 5.32% in the scheduling time, resulted in DMS2 processing 1.54% more jobs than OLB.

To confirm the point of saturation of the systems, we have run a simulation with the same configuration as above, but with the load parameter ranging from 0.1 to 0.9, which means that the average waiting time between groups of jobs ranges from 10 to 1.1 seconds of simulated time.

This way, we can discover if the number of jobs completed per second is kept constant or if there is any increase in decrease on that ratio. The result of this simulation is presented in Figure 58, giving a better view of the difference in performance between the DMS2 and the OLB. From these results, we can confirm that the saturation of both systems happens when the interval between groups of jobs is approximately 11 seconds of simulated time.

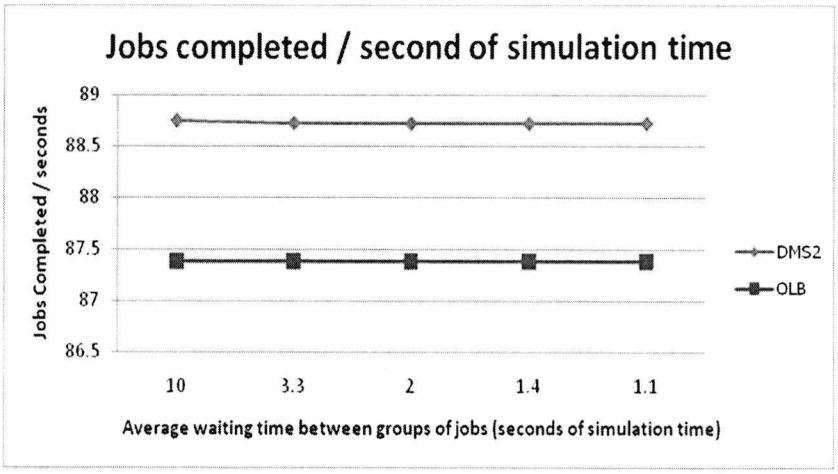

Figure 58 - Jobs completed / seconds of simulated time (load from 0.1 to 0.9): DMS2 vs. OLB.

6.3 Dynamic Scenario

In this scenario, we are interested in evaluating DMS and OLB in a dynamic environment where providers take themselves out of service.

The disconnection of consumers is not considered here, so we can keep the job submission similar to the static scenario and be able to compare both static and dynamic scenarios.

The disconnection of the scheduler is also not considered in this scenario, because we consider this disconnection as a failure, and we do not want to introduce failures to this scenario, and leave simulation of them until the fault tolerance

scenario. For this same reason, we also assume that a provider never turns off while processing a job.

6.3.1 Disconnection Mechanism

The disconnection of the providers is governed by two variables: the connected period (CP) and the disconnected period (DP), and the values for these variables are calculated based on Exponential distributions.

The connected period variable determines the next time that a provider should attempt to turn off after turning on; while the disconnected period variable indicates how long the device should remain disconnected.

The load parameter was already introduced in the static model. Now we need two other parameters for the exponential distributions that are used to calculate the CP and DP variables. These parameters are referred in this thesis as ρ_{CP} and ρ_{DP}, respectively.

The last assumption for this scenario is that there must be at least one provider active in the grid, so we can avoid periods where no jobs are processed, eliminating long idle periods from the results. Therefore, when a provider reaches the time for it to turn off, it only does so if there is at least one other provider on; otherwise, the provider has to wait until another one connects to the grid. This mechanism is improved in the fault tolerant scenario, when all the providers can become unavailable at the same time and the job time-out mechanism can be used and the system does not become fully idle.

6.3.2 Dynamic OLB

The flow of information for this model is similar to the one presented in the static model (see Figure 52), except for the fact that we have had to add a registration message from providers to the scheduler, since the latter needs to know what providers are available. The registration message is sent as soon as a provider becomes ready to accept work items.

One of the disadvantages of OLB is that it is not possible to guarantee the most up-to-date information about the providers, unless the scheduler keeps constantly exchanging messages with providers for this purpose, requiring more bandwidth and also CPU usage to process those messages.

As mentioned in Chapter 3, most of grid schedulers attempt to minimize this problem by making use of a Network Weather Service (NWS), and for the same reasons exposed in section 4.1, a NWS was not implemented in our experiments. Another disadvantage of NWS is that they also add more traffic to the network.

Because of this drawback with OLB, the scheduler may try to send a job to a disconnected provider and eventually a network timeout may occur. Whenever this network timeout happens, the scheduler tries to reallocate the job to another provider.

The scheduler cancels any current attempt to communicate with a disconnected provider when a registration message is received from that provider.

6.3.3 Dynamic DMS2

The flow of information for DMS2 in the dynamic scenario is the same as in the static scenario (see Figure 51). The difference here is in the way that providers and the scheduler behave when a disconnection occur.

In this model, providers never disconnect immediately after sending their local queue to the scheduler; instead, they have to wait until they get a response from the scheduler before they turn off. This decision was made under the assumption that once the provider sends the queue, it becomes committed to processing one of the jobs in the queue, and so it has to wait for a response.

This way, a network timeout never happens for DMS2, which is different from OLB. Later in this thesis we introduce another version of DMS and perform a dynamic simulation where providers can disconnect at any time.

6.3.4 Simulation Set-up

For the results presented in this session, we kept the same simulation set-up used for the results presented for the static scenario. The only difference here is that there are three parameters for the simulation instead of only one: load parameter (λ), ρ_{DF} and ρ_{DP}.

In these experiments, λ is given a constant value throughout the iterations of each simulation, so we can observe the performance of the systems when varying the environment, but keeping a constant load factor.

Only one of the parameters ρ_{CP} or ρ_{DP} has the value changed during the simulation steps, while the other is kept constant, so we can define simulations with specific characteristics changing such as long/short connected periods (CP) and long/short disconnected periods (DP).

6.3.5 Long CPs and long DP: DMS2 vs. OLB

In this section we present the results of running the simulation where we can have long connect periods and providers staying off for a long period.

The values for the parameters of the simulation are summarized in Table 7.

The value for λ was chosen based on the static simulation results, and it is the value for which the systems saturate for that scenario. This way we can keep the high load and verify which one of the systems has its performance degraded more quickly when the availability of providers is reduced.

Table 7 - Parameters for the dynamic simulation: Long CPs and long DP.

Parameter	Value
λ	0.1
ρ_{CP}	0.01 to 0.09 (increment of 0.02)
ρ_{DP}	0.01

By ranging ρ_{CP} from 0.01 to 0.09, we define that the average time between a connection and another disconnection ranges from 100 to 11 seconds (of simulated time). This means that when we increase the value of the parameter, we reduce the availability of providers in the system.

By making ρ_{DP} fixed at 0.01, it means that when a provider becomes unavailable, it stays in that state for an average period of 100 seconds (of simulated time).

The number of jobs completed per second of simulated time is shown in Figure 59. Those results show that DMS2 is able to complete fewer jobs per second than OLB when the availability is reduced in this case.

When consumers broadcast a job description in DMS2, only providers that are available receive the broadcast message and they are the only ones capable of performing the job. When a new provider becomes available, it does not have any mechanism to find out information about those jobs that have been submitted before

and that are still pending execution. The mechanism where providers can retrieve jobs from the scheduler presented in Chapter 4 is not part of DMS2, and it is only introduced in the DMS3, the third version of the DMS described in the next section.

Suppose that the new provider is the only one that is active, and all the others became unavailable. In this case, only newly submitted jobs will be processed by the new provider, while older jobs have to wait for another provider (that knows about them) to reconnect. When the new provider disconnects, the jobs that it received during the period it was the only provider available have to wait for it to come back to the grid. This can also generate unfairness in terms on usage of the system by consumers.

No jobs are lost permanently in this scenario because all providers eventually become available again.

Since a consumer has to wait for all the jobs in the current group to be completed before sending another group, the job submission is delayed when the phenomenon described above happens, making the performance of DMS2 to decrease.

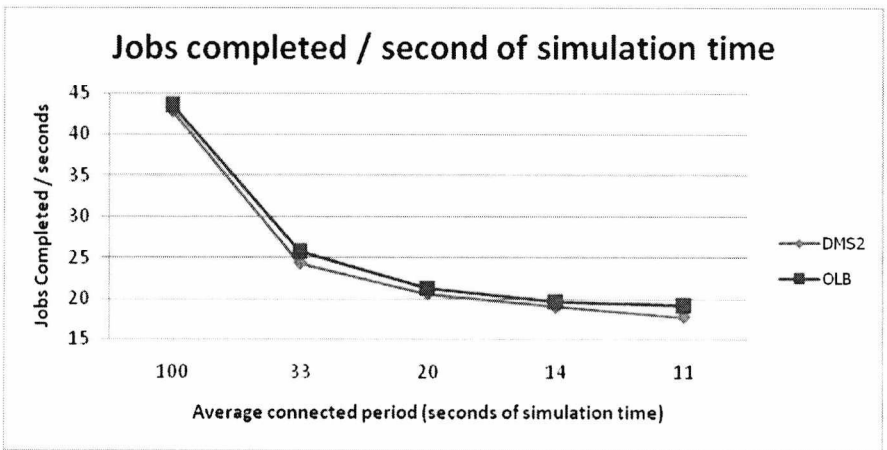

Figure 59 - Jobs completed / second of simulated time: DMS2 vs. OLB (dynamic scenario – long CP and long DP).

The performance degradation of DMS2 becomes more evident when we increase the load and reduce the average connected time. The results in Figure 60 were obtained from a simulation with average time between group submission of approximately 1.1 seconds of simulated time ($\lambda = 0.9$), and average connected time ranging from 10 to 1.1 seconds of simulated time.

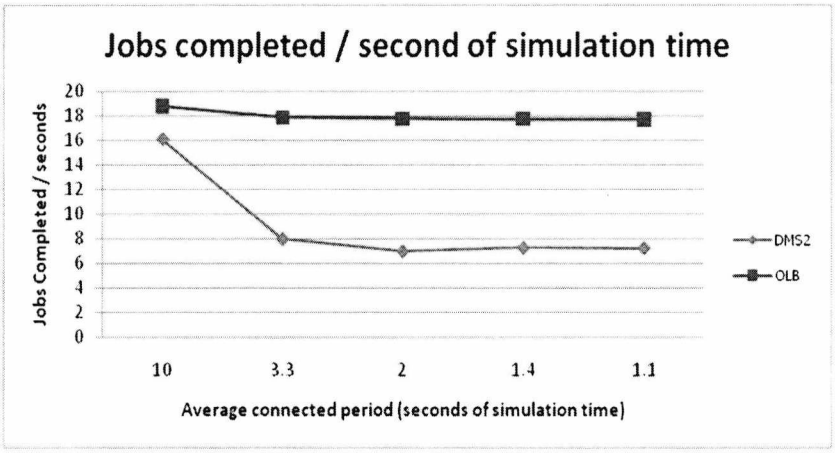

Figure 60 - Jobs completed / second of simulated time: DMS2 vs. OLB (dynamic scenario – short CP and long DP).

While DMS2 has the performance degraded by the reduction of provider's availability, OLB saturates because of high CPU usage on providers.

To improve DMS2's performance, we need a mechanism that allows a provider that just connected to the grid to find those jobs that are pending execution.

6.3.6 DMS version 3

Although DMS2 has shown good performance compared to OLB in the static scenario, we have seen that DMS2's performance might not be good in the dynamic scenario where providers become unavailable.

In the third version of DMS (DMS3), we have added new information exchanges to the protocol in order to make it possible for newly connected providers to retrieve information about pending jobs in the scheduler. DMS3 also anticipates the fault tolerance mechanisms (which are not present in DMS2 or OLB), which will allow the recovery of the scheduler's status.

For DMS3, we needed a way to specify the order of arrival of the jobs at the scheduler. Since we do not use timestamps (as mentioned in Chapter 4), we decided to use a simple mechanism where the scheduler allocates a numerical ID to the jobs and sends this ID back to the consumer during the job submission process.

DMS3 corresponds to the scheduling system presented in the section 4.4 of this thesis. This is the version where providers send the information about possible missed jobs (oldest off interval) to the scheduler, along with the queue of jobs they are able to process (Pq); the scheduler then replies with the selected job (if any), the

updated Pq (Pq'), the list of jobs that have been missed by the provider (Ls) and a flag indicating if there are still more jobs that have been missed by the provider ("more flag". Please refer to section 4.4 for the full description of DMS3.

The complete flow of information simulated for DMS3 is the same one presented in Figure 23 (Chapter 4).

6.3.7 DMS3 vs. OLB

This session presents the results of simulation DMS3 in the dynamic scenario, using the same configuration applied to the simulation of DMS2.

The number of jobs completed per second of simulated time is presented in Figure 61. These results correspond to the simulation with the parameters in Table 7.

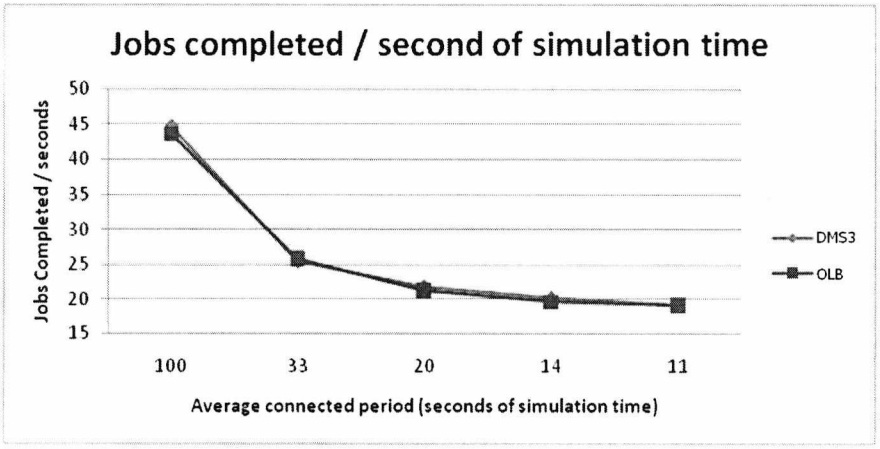

Figure 61 - Jobs completed per second of simulated time: DMS3 vs. OLB (dynamic scenario – long CP and long DP).

These results show that DMS3 and OLB perform similarly (with slightly better performance) in this configuration. The difference between these two systems becomes more emphatic when we reduce the availability of providers by decreasing the average connected period and keeping the same disconnected period (see Figure 62).

These results show that DMS3 scales better than OLB in this case. We have also run the simulation with different values for λ, ρ_{CP} and ρ_{DP} (although keeping the same job size used so far) and the results were similar to the ones presented in Figure 62.

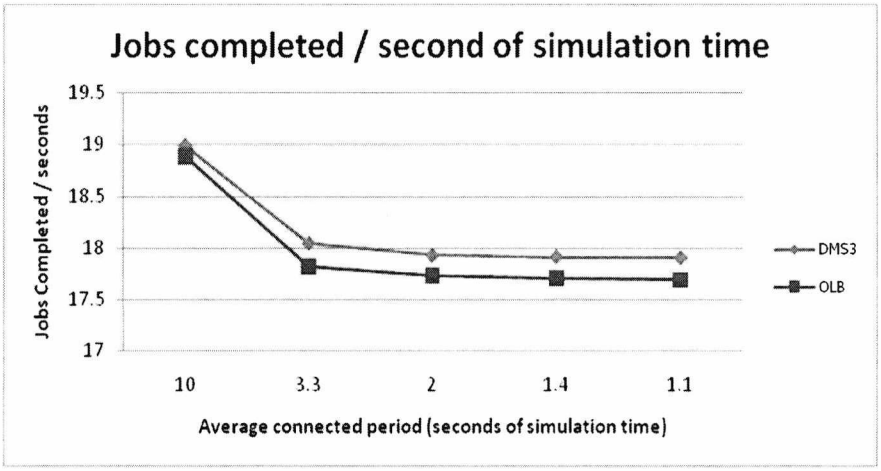

Figure 62 - Jobs completed: DMS3 vs. OLB (dynamic scenario – short CP and long DP).

6.3.8 Static scenario: DMS3 vs. DMS2

In this section, we have the comparison of results between DMS3 and DMS2 to check if the extra processing and network usage required by DMS3 affects the performance.

Since there are no disconnections in the static scenario, the information about the oldest off interval is always "null" in the simulations of this scenario. For the same reason, the list of missed jobs sent by the scheduler to providers is always empty, and the "more flag" is set as false for all simulations in this scenario.

Because of this, DMS3 does not add much processing to the system when compared to DMS2 in the static scenario. Most of the impact of applying DMS3 to this scenario is seen in the network usage, as we can see in Figure 63.

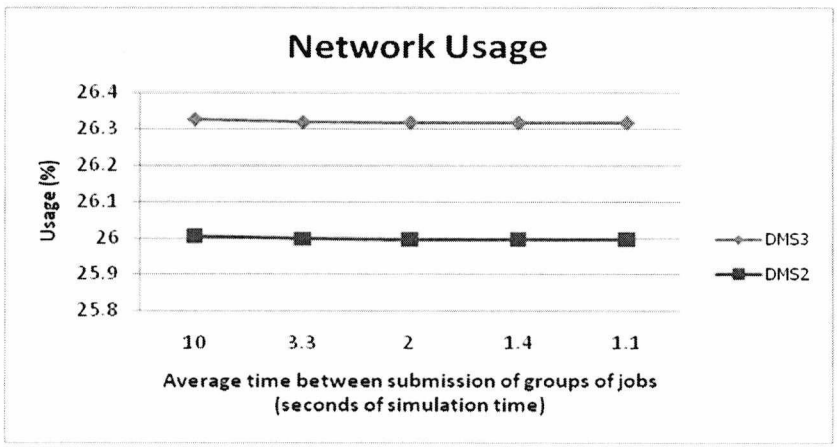

Figure 63 - Network Usage: DMS3 vs. DMS2 (Static Scenario).

This network usage is from a simulation with the same configuration presented earlier for DMS1 and DMS2. The load parameter ranges from 0.1 to 0.9.

The 0.3% (approximately) of difference in network usage between DMS3 and DMS2, shown in Figure 63, results in DMS3 processing approximately 0.03% fewer jobs than DMS2 (at load parameter 0.9), as we can see in Figure 67.

Despite the small loss of performance in relation to DMS2, DMS3 still completes more jobs than OLB.

6.3.9 Impact of the *Job List* size

The results presented in the other sections of this thesis use $N = 10$, where N is the size of the list sent by providers to the scheduler. As we have shown earlier, a big size of that list can considerably degrade the performance, and for this reason, we limited that size when we created DMS2.

In order to decide a good value for N, we ran some DMS3 simulation setups with different N values. Results have shown that DMS3 performs better when N is close to double of the number of providers.

For example, the results in Figure 64, presents the number of jobs completed per simulated time considering the same configuration used for the results in Figure 58 (static simulation with 100 consumer, 5 providers and time between groups of jobs ranging from 10 to 1.1 seconds).

The values used for N were 1, 5, 10, 20 and 30. N=10 presented the best results, while N=1 presented considerably poorer performance when compared to the remaining values, which happens because the update the provider's local queues take longer to be updated with the submission of only 1 element at a time to the scheduler.

Figure 65 shows the curve where the performance starts to improve, reaches a limit, and starts to degrade. For this curve, we considered the number of jobs completed per simulated time from Figure 64 when the time between groups is 1.1 seconds.

The results in Figure 65 arbitrarily use an average time of 0.25 between the submissions of jobs within the same group. When we increase the load of the system by reducing this average time to 0.005, the performance curve becomes more emphatic, and shows poor performance when $N = 5$, as shown in Figure 66.

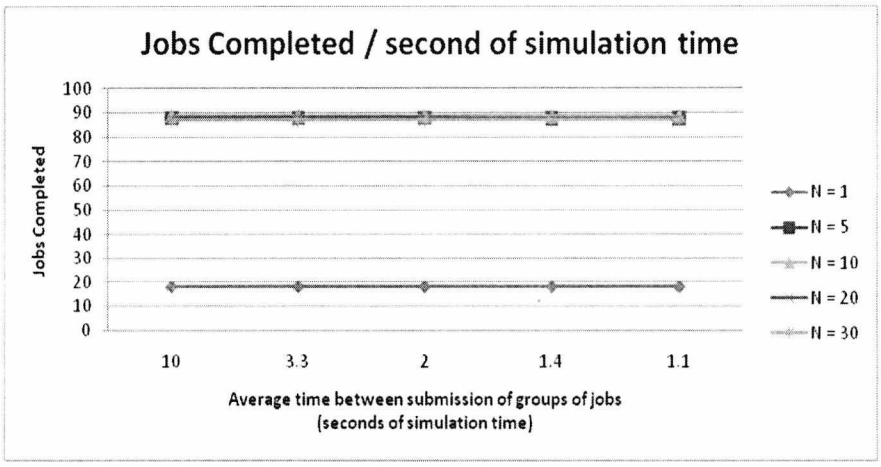

Figure 64 - Impact of the job list size.

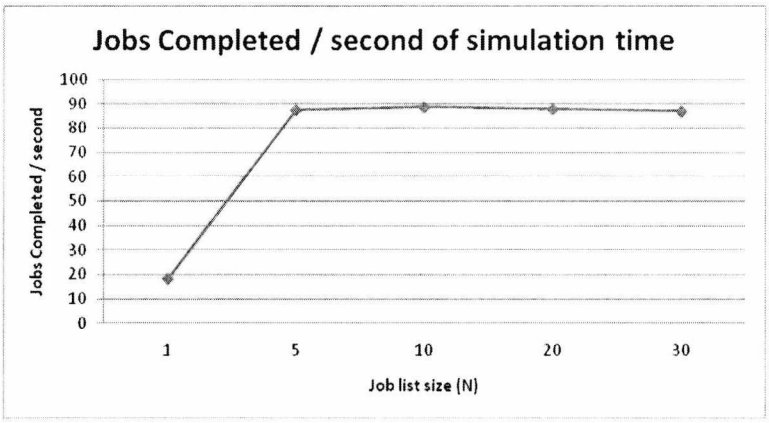

Figure 65 - Performance curve for the impact of the job list size.

Considering the importance of the size of the list, an addition to the future version of the protocol would be for the scheduler to inform the providers about the size of the lists that they should send, based on the number of providers registered in the grid.

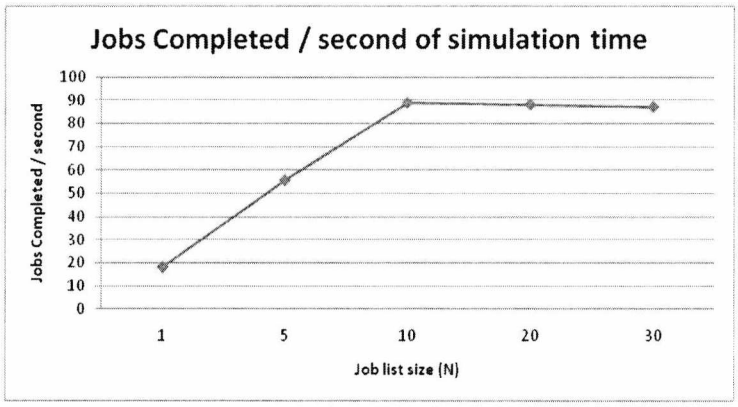

Figure 66 - Performance curve for the impact of the job list size: higher load.

6.3.10 Impact of the Network Usage

One of the main worries during the design of our scheduler was to reduce the network usage, since we believed that this could affect the overall performance of the grid system.

In order to verify how much a higher load on the network can degrade the performance of the grid system, we have run the static scenario simulation with the same configuration used in the results presented so far, except for the fact that the executable file is 20 KB in this simulation, instead of the previous 1 KB. The load parameter ranges from 0.01 to 0.09 (submission time from 100 to 11 seconds of simulated time).

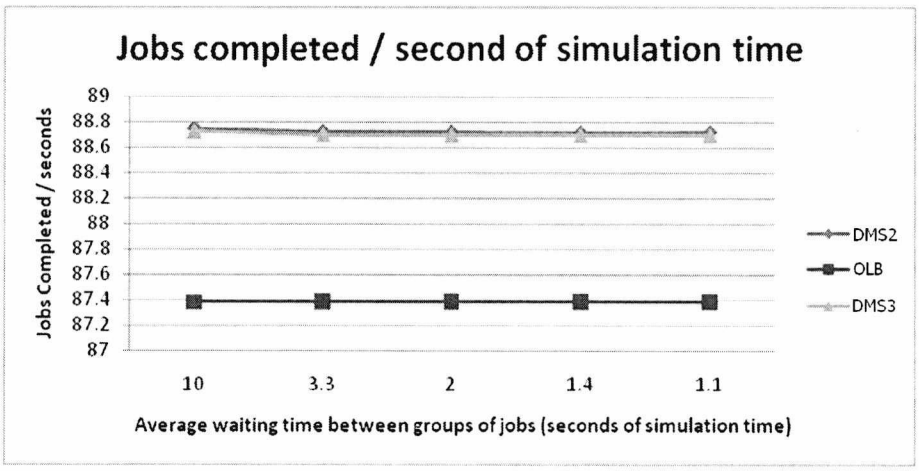

Figure 67 - Jobs Completed: DMS3 vs. DMS2 (Static Scenario).

With this experiment, we expect an increased delay of the messages related to the grid system, and, consequently, a decrease in the number of jobs completed.

The graphs presented in Figure 68 and Figure 69 show the resulting network usage for DMS3 and for OLB when using 1 KB and 20 KB of executable file, respectively.

These results show a considerable increase of the network usage. In the results presented in Figure 69, we can see that the use of the network is almost 100% and it saturates when the time between groups of jobs is 33 seconds of simulated time (which corresponds to 0.03 as load parameter). We can also see that, in both cases, the network usage is similar for DMS3 and OLB.

As we can see from Figure 55, when using 1 KB for the executable file and the average time between submissions of groups of jobs ranging from 100 to 11 seconds of simulated time, OLB saturates when the time is 14 seconds, and produces approximately 88 jobs per second of simulated time. However, when we increase the executable file to 20 KB, OLB saturates when the average inter-submissions time is 33 seconds, completing approximately 30 jobs per second of simulated time, as we can see in the graph represented by Figure 70.

These results show that the network load can have a big impact on the total performance of the system in terms of number of completed jobs. For this reason, we have to avoid any unnecessary usage of the network by the scheduling protocol itself.

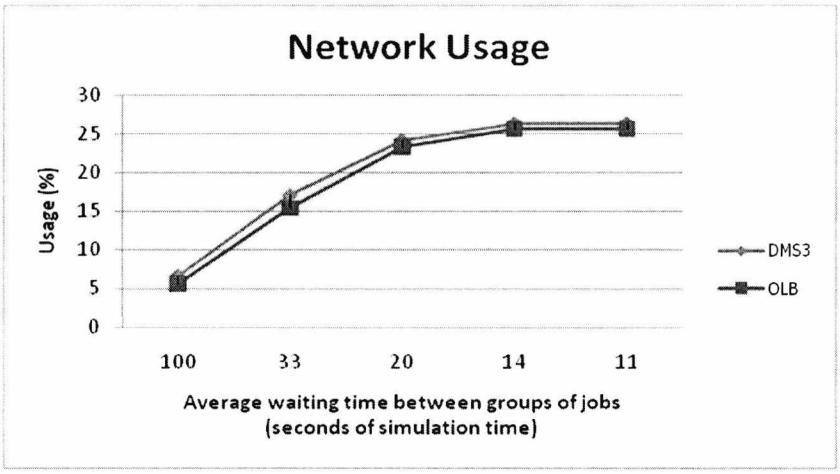

Figure 68 - Network Usage: DMS3 vs. OLB (1KB of executable file).

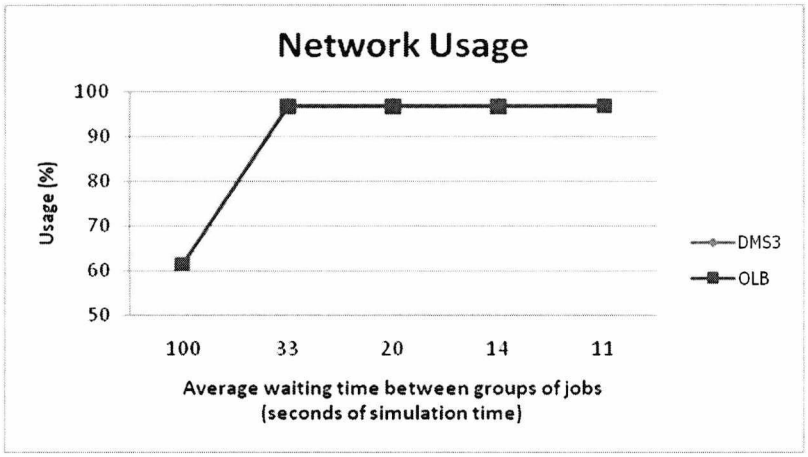

Figure 69 - Network Usage: DMS3 vs. OLB (20 KB of executable file).

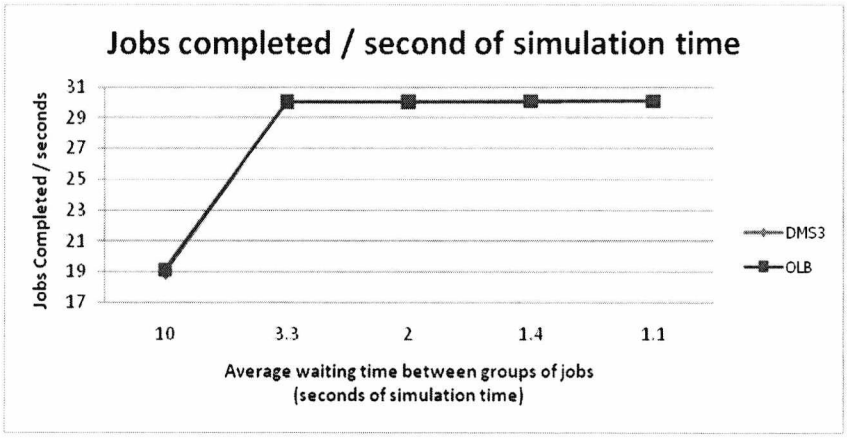

Figure 70 - Jobs completed: DMS3 vs. OLB (20 KB of executable file and load from 0.01 to 0.09).

6.3.11 Scalability on the number of providers: DMS3 vs. OLB

So far we have been presenting results for the dynamic scenario with the assumption that at least one provider is active in the grid, which can generate periods in the simulation where there is only one provider working.

OLB is expected to perform better with fewer providers active in the grid, since it has complexity on the order of $O(m)$, where m is the number of available providers. For this reason, in this section we summarize the results of simulating OLB and DMS3 with different values for the minimal number of active providers (P_{min}).

The simulations were performed with the same configuration used to obtain the results presented in Figure 60 ($\lambda = 0.9$, ρ_{CP} ranging from 0.1 to 0.9 and $\rho_{DP} = 0.01$),

but with different values for P_{min}, which ranges from 1 to 5 (the latter corresponds to a static environment in this case where we have 5 providers at most).

The graph in Figure 71 represents the difference between the number of jobs per second of simulated time produced by DMS3 and the total produced by OLB when considering an average CP of 1.1 seconds of simulated time ($\rho_{CP} = 0.9$).

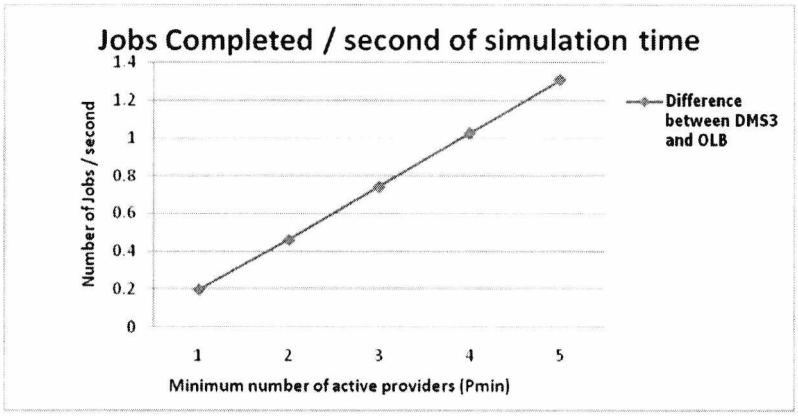

Figure 71 - DMS3's scalability when increasing the number of providers.

Even though the difference is not very big in these results, they show that DMS3 scales better than OLB with the increase of the number of active providers in the grid.

6.3.12 Short CP and Short DP: DMS3 vs. OLB

In the previous sections, we have discussed the results considering long disconnected periods for the providers. In this section, we present the results of a simulation with short disconnected periods and short connected periods, representing a very dynamic environment with many disconnections from providers.

To achieve short DP, we gave ρ_{DP} a constant value of 0.9 throughout the simulation, which makes the average disconnected period to be approximately 1.1 seconds of simulated time. With regard to the CP, ρ_{DP} ranges from 0.1 to 0.9, corresponding to approximate average CP of 10 to 1.1 seconds of simulated time, respectively.

The number of jobs produced by the simulation of this configuration is presented in Figure 72 and once more DMS3 produces slightly better results than OLB. It was assumed $P_{min} = 1$.

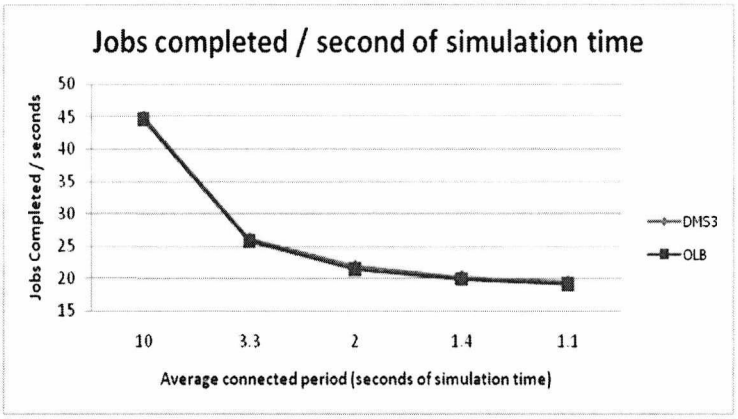

Figure 72 - Jobs completed: DMS3 vs. OLB (short CP and short DP – P_{min} = 1).

6.4 Fault Tolerance Scenario

So far, we have shown results for simulations where none of the devices disconnect and cases where only providers become available. In this section, we explore some of the fault tolerance mechanisms specified in Chapter 4, such as the timeout for job completion, the election of a new scheduler and the recovery of its status.

6.4.1 Election's Utility Function

One of the extra mechanisms added to our fault tolerant system is the election of a new scheduler, which uses the utility function described in section 4.5. One of the decisions taken for that function was that higher processing power should contribute positively to the utility value of a device during the election. In this section, we describe the experiment that was set in order to support that decision, and its respective results.

The experiment considers two cases: one with the scheduler component running alone (i.e., without sharing the processor with a consumer or provider component) in a device with limited processing capabilities; and another with the scheduler running on the same device as a provider, assuming the devices with higher capabilities work as providers.

The results in Figure 73 show the amount of jobs produced after simulating the static scenario considering 100 consumers and 5 providers, and the load parameter ranging from 0.1 to 0.9 (i.e., the average time between submission of groups of jobs varies from 10 to 1.1 seconds of simulated time). We chose the static scenario for this experiment because it keeps all the providers connected to the grid, increasing

the load on the scheduler with the scheduling requests, which provides better comparison between the two cases.

In order to consider only the overhead caused by the scheduling related processes, we have set the job sizes to 0 (zero), so that providers do not spend time with the execution of the jobs. The data transferred on the network was also set to the minimum necessary to have jobs submitted, scheduled and completed, excluding I/O data.

For all the results presented so far, we have used 0.5 as the parameter for the Uniform distribution that defines the time interval between the submissions of jobs in the same group, which means that this interval ranges from 0 to 0.5 seconds of simulated time. When using 0.5, the simulations do not reach a saturation point, making it difficult to compare both simulations. For this reason, we reduced the parameter to 0.01, making the interval ranges from 0 to 0.01 seconds of simulated time.

The "Powerful device" series represents the number of jobs completed per second when the scheduler shares the processor with a provider, whereas the "Limited device" series represents the amount of jobs processed when the scheduler runs alone (without sharing the processor with a provider or consumer) in a limited device.

The results in Figure 73 show better performance for when the scheduler shares processor with a provider, showing an improvement of 8%, approximately. One of the reasons for this improvement in performance is the difference on the scheduling time (see Figure 74), which is due to less CPU usage (see Figure 75) in the device with faster processor.

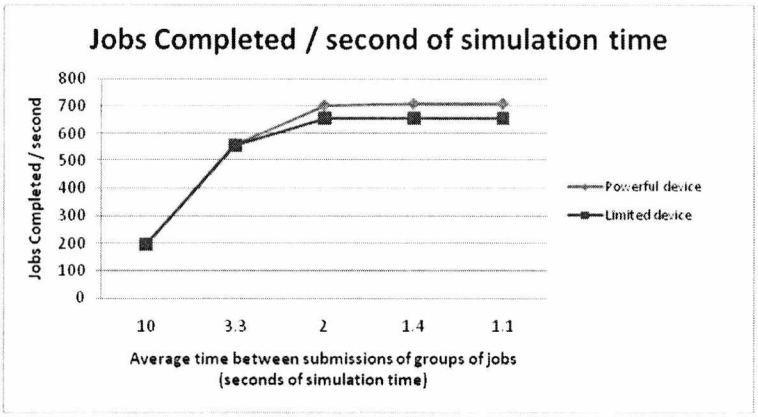

Figure 73 - Jobs completed: scheduler on powerful device and on limited device.

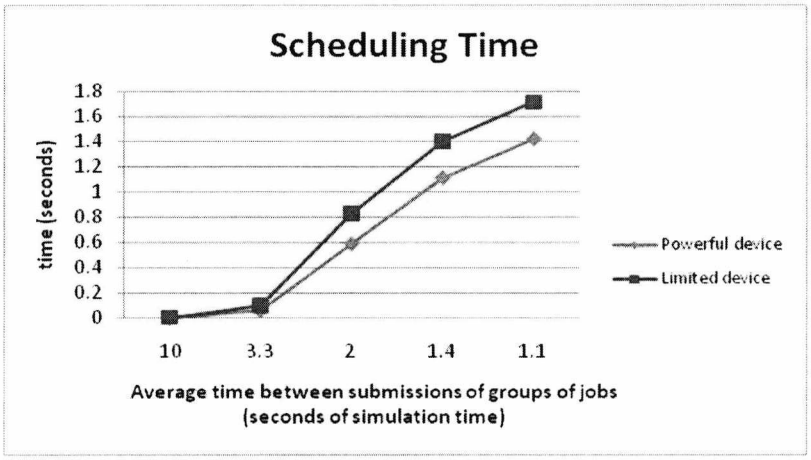

Figure 74 - Scheduling time comparison: scheduler on powerful device and on limited device.

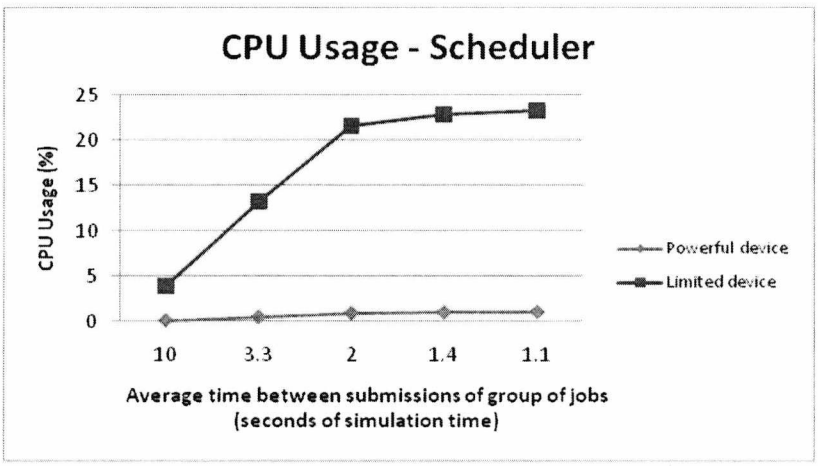

Figure 75 - Scheduler's CPU usage: scheduler on powerful device and on limited device.

Even though the CPU usage is much higher in the "Limited Device" case, it has small impact in the scheduling time, since most of the time is spent with communication (near 100% of network usage), which is a bottleneck for this kind of system, as mentioned earlier in this thesis. Figure 76 shows the network usage for this simulation. The results show that the scenario with a powerful device as scheduler uses more network than the one with a limited device, which is due to the higher number of jobs processed in the first case.

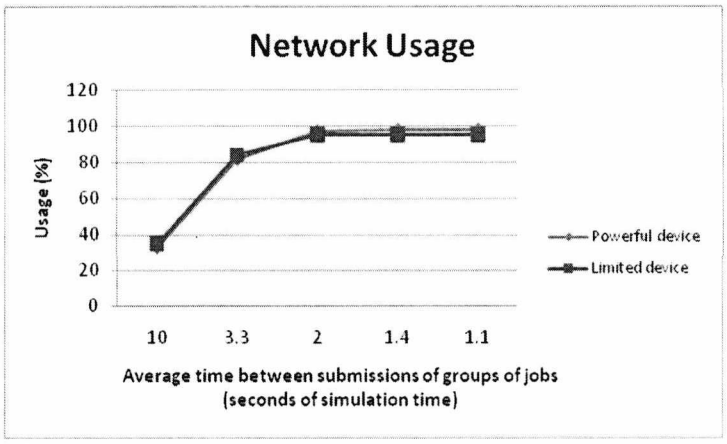

Figure 76 - Network usage comparison: scheduler on powerful device and on limited device.

6.4.2 Job completion timeouts

In DMS3, we have a mechanism where providers can find out about the jobs submitted during their disconnected periods. That mechanism works well when the scheduler is always available, since the scheduler will always know about the status of all the submitted jobs.

When we consider a dynamic environment with the scheduler becoming unavailable, it may happen that the recovery process of the new elected scheduler described in Chapter 4 does not recover all the jobs, since some jobs may not be known about by the providers that are available during the recovery.

For this reason, we have introduced timeouts for the jobs, where consumers have to send a reminder of the jobs that were not completed within a certain amount of time T.

In this section, we evaluate the impact of the timeouts in the number of jobs completed. At this stage, we present the results of simulating the dynamic scenario considering long disconnected periods (average of 100 seconds of simulated time) and the introduction of job completion timeouts, so we can verify if they add much processing and communication overhead to the system. We have also increased the load parameter to 0.9 (approximately 1.1 seconds of simulated time between group submissions), since it increase the possibility of a job timeout happening with more jobs submitted to the system.

In the next section, we evaluate the impact of these timeouts when the scheduler becomes unavailable.

The results in Figure 77 present the number of jobs completed when there is no job completion timeout and when we introduce timeouts of 20 and 60 seconds of simulated time.

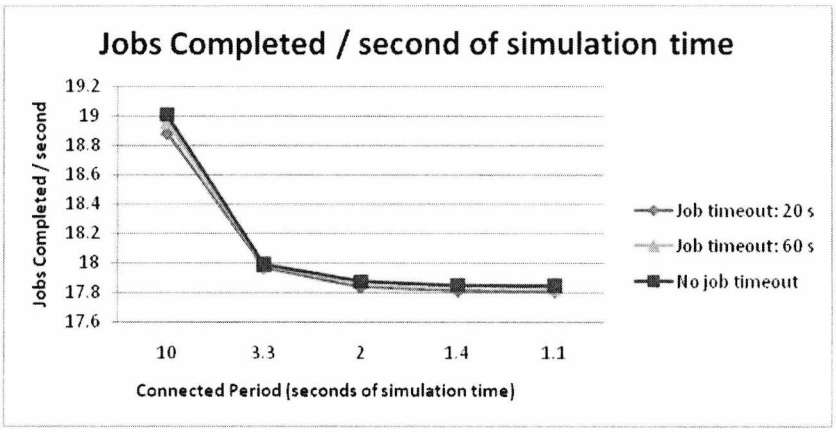

Figure 77 - Job completion considering job's timeouts.

These results show that, for the "dynamic scenario", the presence of timeouts for the jobs do not improve the performance of the system, but produce slightly worse performance than having no timeout at all.

In Figure 78, we can see the number of retrial messages (messages broadcast by a consumer after it gets a timeout for the job) considering T=20 and T=60 seconds of simulated time. A smaller timeout contributes considerably to an increased number of retrials and, consequently to the increase of number of messages transmitted on the network.

Even though there can be a small increase in network usage, we can conclude that, for the static scenario, the introduction of job completion timeouts does not have much effect on the number of jobs completed, since the jobs will probably be completed before the timeout in that case.

The decision of using timeouts for the jobs is up to the application, which can also decide the length of the timeout and how many times a reminder can be sent for each job. In our experiments, we did not set any bound on the number of retrials, assuming that this would be the worst-case scenario, with a high network usage.

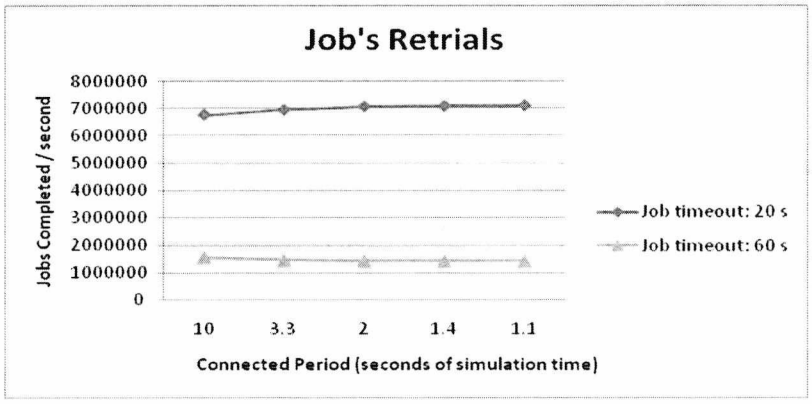

Figure 78 - Job's retrials considering job's timeouts.

6.4.3 Scheduler's failure: DMS version 4

Here we evaluate the performance of the system when the scheduler fails and then a new one has to be elected and recovered.

Since OLB does not specify fault tolerance mechanisms, we only compare the results with DMS3, so we can verify how the performance degrades when the scheduler starts to fail. In this thesis, we refer to the simulation of DMS considering fault tolerance mechanisms as DMS4.

The disconnection of devices follows the same procedure used for the dynamic scenario, where we have to specify a parameter for the connected period and another for the disconnected period. Consumers do not disconnect in order to keep the load similar to that used in the other scenarios.

Once the device that runs the scheduler disconnects, the election of a new scheduler starts as soon as another component (registration, consumer or provider) reaches a connection timeout after sending a message to the scheduler.

For the election, only the original limited device running the scheduler and those devices running a provider have been defined as scheduler capable (those that can run the scheduler and, consequently, are candidates during the election).

In order to simplify the simulation, the candidates compute a random utility value every time the election starts, so all the candidates may have the chance to run the scheduler.

As specified in Chapter 4, both election and recovery processes have to wait for a certain amount of time ($T_{election}$ and $T_{recovery}$, respectively) which determines its completion. For this reason, two new parameters where introduced to the simulation to represent $T_{election}$ and $T_{recovery}$.

Before the start of the election process, consumers and providers are paused and they are resumed after the recovery process. This pause of those componentss reduces the usage of the network, allowing the exchange of messages during the election and recovery to be performed more quickly. For this reason, $T_{election}$ does not need to be a high value.

For the results presented in Figure 46, we have used $T_{election}$ = 1 second of simulated time (although we have observed that the election in fact takes less than 2 milliseconds of simulated time, considering the configurations that we have been simulating).

The results in Figure 79 were acquired after running the simulation using the values presented in Table 8.

As it was expected, DMS4 produces fewer jobs than DMS3, since the device running the scheduler never becomes unavailable.

In order to make a fairer comparison, DMS3 uses a powerful device to run the scheduler (sharing processing power with a provider).

In DMS4, all the devices have the same probability of becoming unavailable, without a "reliable" device to run the scheduler, which goes against one of the assumptions for our system, but here we are interested on evaluating the performance of the system in extreme situations, where the scheduler becomes unavailable very often and many election processes happen.

Table 8 - Configuration used for the simulation of the fault tolerance scenario.

Number of consumers	100
Number of providers	5
$T_{recovery}$	2 seconds of simulated time
$T_{election}$	1 second of simulated time
Load parameter	0.1 (average of 10 seconds of simulated time between group submission)
Connected period	Ranging from an average of 100 to 11 seconds of simulated time
Disconnected period	Average of 100 seconds of simulated time
Job completion timeout (T_{CT})	60 seconds of simulated time

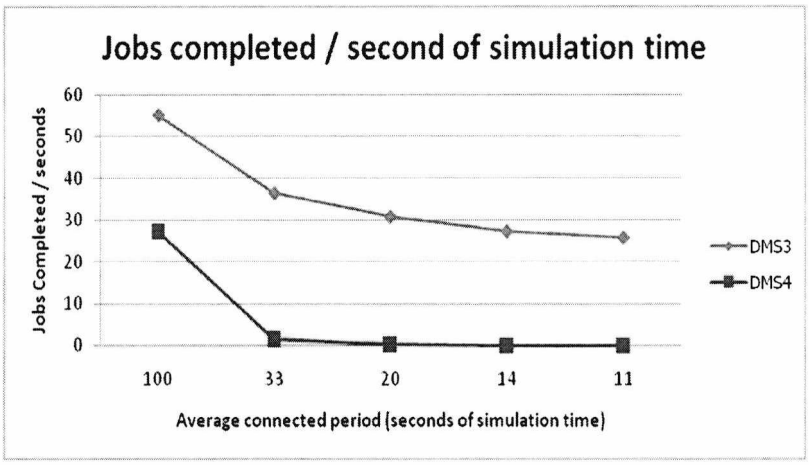

Figure 79 - Job completion: DMS3 vs. DMS4 (T_{CT} = 60 s).

Note that DMS4 depends on failed communications with the scheduler in order to identify the scheduler's failure and start an election. Since the load for this simulation is not high, it generates longer idle periods for consumers, and, consequently, for providers, where these components do not attempt to communicate with the scheduler, taking longer to identify a failure and start an election. When they attempt any communication with the scheduler, they also have to wait for the network timeout in order to identify the failure.

Associating this fact with the low availability of devices, the system can stay idle for long periods because of not having an active scheduler, causing the performance of DMS4 to fall considerably when compared to DMS3.

Another fact that also contributes to the reduction in the number of processed jobs is that nothing is produced during the election and recovery periods.

Since the dynamic scenario produces much better results when compared with our fault tolerant scenario, we decided not to continue with this kind of comparison and compare two fault tolerant systems instead; one with and another without recovery process.

6.4.4 Scheduler's failure: Recovery vs. No Recovery

In this section, we analyze the performance of running the system without recovering the scheduler's queue after the election process. Instead, all consumers have to resubmit the jobs that have not been completed before the scheduler's crash.

To simplify the explanation of the results, this alternative version of our system will be referred as DMS5.

For this scenario, we assume that providers do not empty their queues when they become unavailable, so they do not have to perform the matchmaking process again for the same job. Thus, to avoid generating an inconsistent state of the providers, the recovery of the global ID is performed similarly to that in the simulation presented in the previous section.

The results presented in Figure 80 show the number of jobs completed per second of simulated time when running the simulation with the configuration presented in Table 9.

We have set a fixed parameter for the disconnect period in order to have an average time that is longer than the network timeout, so, once the scheduler crashes, the devices communicating with it will identify the failure.

Here we decided to increase the connected period and the load, so we can augment the number of successful elections and recoveries, and evaluate the impact of these processes in the overall result.

Table 9 - Set-up used for the simulation of the fault tolerance scenario considering no recovery of the scheduler's queue.

Number of consumers	100
Number of providers	5
$T_{recovery}$ (seconds of simulated time)	2 seconds for DMS4; and 1 second for DMS5
$T_{election}$ (seconds of simulated time)	1 second
Load parameter	0.9 (average of 1.1 seconds of simulated time between group submission)
Disconnected period	Average of 100 seconds of simulated time
Connected period	Ranging from an average of 1000 to 111 seconds of simulated time
Job completion timeout (T_{CT})	60 seconds of simulated time

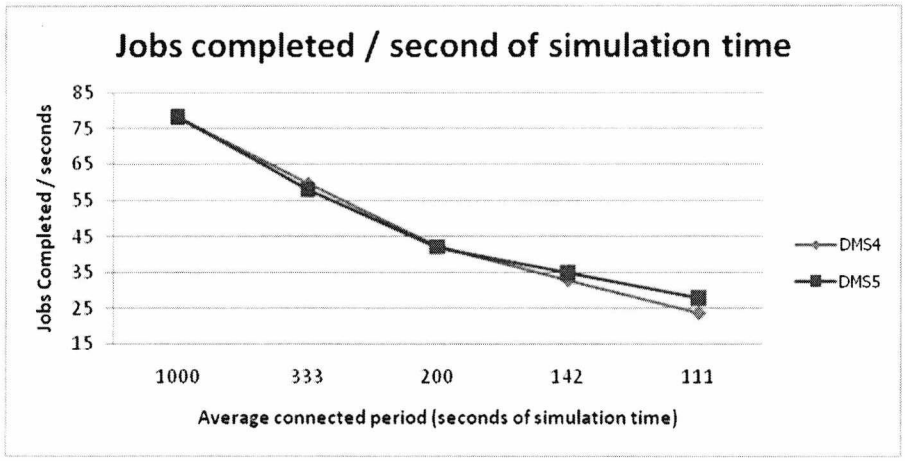

Figure 80 - Jobs completed per second: Recovery vs. No recovery.

These results show that DMS5 can perform better then DMS4, which is expected, considering that DMS4 has at least 1 extra second of simulated time without jobs being processed during the recovery process. At the last point of the graph (111), about 220 elections happened, which means that there were at least 220 seconds of simulated time that DMS4 did not use for job processing.

DMS5 performance is limited by the resubmission of jobs from consumers after the recovery process, requiring high network usage (bottleneck), which is not good for limited devices running on battery power. In DMS4, the scheduler attempts to recover its queue by communicating only with providers, saving battery of consumers.

On the other hand, DMS4 may have a high number of retrials for jobs that have reached the completion timeout. Results have shown a reduction in the number of retrials by augmenting the size of the queue of completed jobs that providers send to the newly elected scheduler during the recovery process. This permits it to recover a more accurate state of the scheduler and avoid unnecessary messages from providers with already completed jobs in the queues sent with the scheduling request to the scheduler, saving bandwidth.

Considering that the participation of consumers in the recovery process produces a more accurate recovered state of the scheduler's queue, we can envisage a version of DMS4 where the participation of consumers and providers in the recovery process can be configured in a local policy by the administrator or by the device's owner.

6.5 Discussion

In this chapter, we have shown the evolution of our solution and its performance on different scenarios against OLB. In most of the cases, the final version of our solution showed better performance than OLB, being similar in others, with the advantage of our solution providing fault tolerance.

Due to time constraints, we were not able to explore more scenarios for this thesis, leaving them for future work (see Chapter 8).

In our simulations, we have considered that, for simplicity reasons, the queues of completed jobs held by providers have a pre-defined maximum size, and when a new job needs to be added to it after reaching the limit, the oldest one in it is removed first in order to keep its maximum size.

These queues are important in the process of updating the scheduler's queue after the recovery because they can reduce the communication between providers and the scheduler to produce a successful assignment of jobs; and their sizes may increase the communication and time for the recovery process.

Considering this, more study is necessary to determine the best way of clearing up the queues of processed jobs. For example, maybe creating daemon processes to remove jobs from them after a certain period, or asking the provider to send information about the oldest job to the providers along with the scheduled job, so that they can perform better updates on the queues.

Chapter 7. Model Checking Results

7 Model Checking Results

In this chapter, we present the models for two cases of network partition and the behaviour of our system in those conditions.

The cases presented in this chapter have not been checked with the simulations. We decided to use model checking because we would not be able to compare our results with OLB, since it does not provide any fault tolerance for network partitions, and because building a simulation would demand much more time to implement it and to test it to make sure that it provides correct results.

Model checking also gives us the guarantee that all the possibilities have been verified and that there are no deadlocks, which is not possible with simulations.

Before we describe the models, first we briefly introduce UPPAAL [109], which is a tool for modelling, validation and verification of real-time systems as networks of timed automata, extended with data types (bounded integers, arrays, etc.).

7.1 UPPAAL

UPPAAL's specification language provides many features which facilitate the description of concurrent real time distributed systems. Here we offer a brief account of the model (please refer to [110], and UPPAAL's help files, for a more detailed presentation).

UPPAAL automata are finite state machines (locations and edges), augmented with clock and data variables, and synchronisation primitives. Concurrent systems are represented by networks of communicating automata. Concurrency is modelled by interleaving, and communication is achieved by synchronisation on channels, or by shared variables. Clocks range in the non-negative reals and advance synchronously at the same rate (but may be updated independently).

Edges denote instantaneous actions, and delays are possible only in locations. Clock and data variables can be used to constrain the execution of automata. Locations may be annotated with invariants, which constrain the allowed delays. Edges may be annotated with guards (enabling conditions), synchronisation labels (to distinguish observable from internal actions), and variable updates. Binary

channels are blocking: matching input and output actions may only occur in pairs (a?/a!). More elaborate specifications can be obtained with the following features.

- **Variables:** Available types include clocks, channels, bounded integers and Booleans, and arrays and record types can be defined over these types. Common arithmetic operators (and user-defined C-like functions) may be used in expressions. Clock constraints are (in)equalities between clocks (and clock differences) and integer expressions. Clocks can be assigned non-negative integer expressions;
- **Urgent and Committed locations:** Urgent and committed locations disallow delays, forcing the immediate execution of enabled actions as soon as they are entered. In addition, committed locations restrict interleaving: only components that are currently in committed locations may execute enabled actions. Figure 81 shows the notation of Urgent (a) and Committed (b) locations;

(a) (b)

Figure 81 - Representation of Uppaal's Urgent and Committed locations.

- **Urgent and Broadcast Channels:** Synchronisation on urgent channels (declared in Uppaal as *chan <name of the channel>*) is binary, blocking, and must occur as soon as matching actions are enabled. Synchronisation on broadcast channels (declared as *broadcast chan <name of the channel>*) matches one output action with multiple input actions, and is non-blocking on the output side: input actions block until the output action is enabled, but the output action may be executed even if no input actions are enabled;
- **Templates and Selections:** Parametric templates and selections provide a concise specification of similar components. A template provides an automaton and a number of parameters (bounded data variables), which can be read in the automaton's expressions (e.g., guards). Parameters are instantiated, generating multiple processes with the same control structure (the template's automaton). A selection denotes non-deterministic bindings between an identifier and values in a given range. Selections annotate edges,

which may use the identifiers in guards, synchronisation labels and updates. Every binding results in a different (instantiated) edge between the same two locations.

7.2 Scenario 1: Simple partition

In this section, we present the case where a network partition happens and one of the sub-networks contains only consumers that are not set as scheduler capable.

In this case, assuming that the partition occurs for a period that is long enough for consumers to identify the absence of a scheduler and start an election, the expected behaviour is that those isolated consumers fail to elect a new scheduler (since none of them is set as scheduler capable). They then wait until the merge of the partition is detected (the merge detection is described in Chapter 4), and they can be part of the grid again.

If the partition merges before the consumers identify the absence of the scheduler, those consumers will act as if the partition had not occurred.

Even though UPPAAL supports the definition of timed automata, we opted for not making use of this facility, and decided to make a model that is generic enough to verify both cases (long and short partition period), without considering time.

This model allows us not only to verify if the system is tolerant to network partition, but also allows us to check the behaviour of the system when consumers become unavailable, which was not verified with the simulations, since we had to keep the same load of jobs in order to have a better comparison between systems and scenarios.

In order to reduce the state space to be explored during verification, some of the behaviours that have been verified with the simulation were not added to the model. For example, we did not model the disconnection of providers in this case.

The remainder of this section consists of the description of the models for the consumer, provider and scheduler. We also describe how the partition was modelled.

7.2.1 Consumer Template

Figure 82 shows the states and transitions that represent the behaviour of consumers.

The *Idle* state is the one where the consumer is not performing any processing related to the grid system. *GetId* represents the state where the consumer waits for a message from the scheduler with the global ID for the last job that has been submitted. At the *GetId* state, the consumer can also receive completion messages from providers (represented by the channel *jobcomp* in the model).

The *Bcast* state is the one where consumers broadcast the job description to providers via the *bcastjob* channel.

The transition from *Idle* to *Wait* happens when a long partition happens, with time for the consumer to identify the absence of the scheduler and attempt election. The channel *whole* indicated that the partition has merged, and it models the behaviour where consumers detects the merge and can (re)submit tasks and receive the job completion messages.

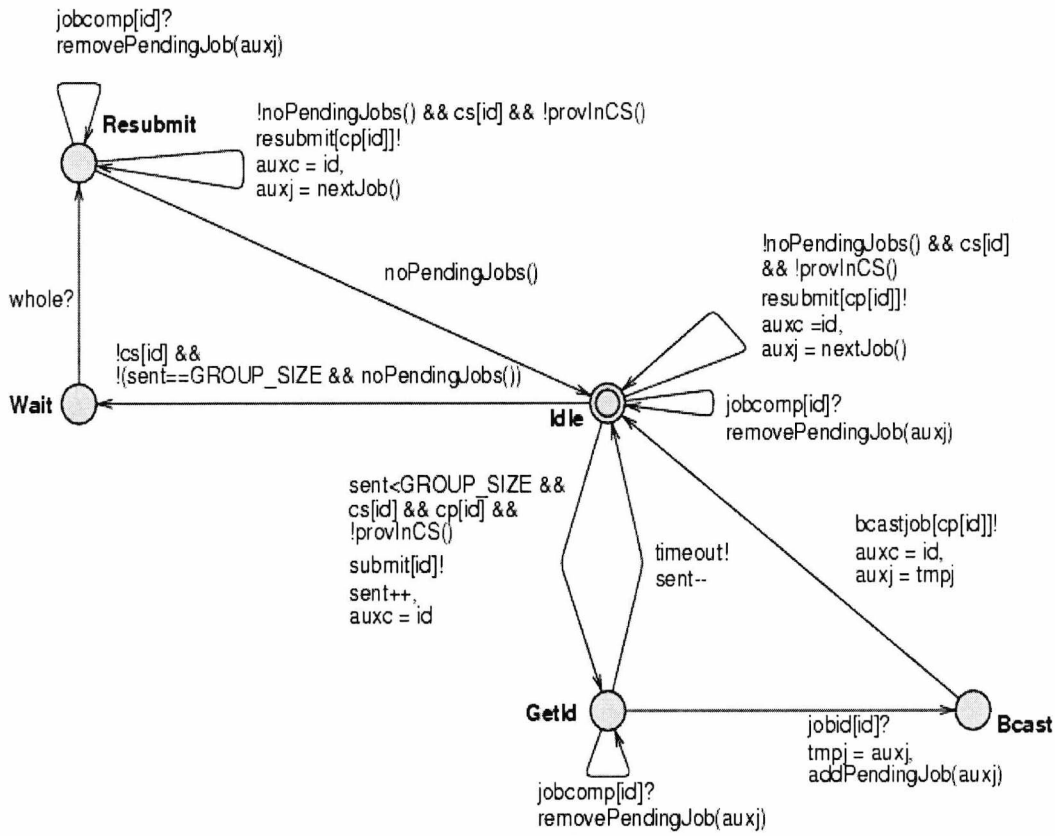

Figure 82 - Simple partition scenario: model of consumers' behaviour.

Here we also assume that jobs are submitted in groups of a fixed size (represented by the variable *GROUP_SIZE* in the model), similar to the simulation models presented earlier in this thesis. The variable *sent* in this template stores the

number of jobs submitted, so that the consumer stops submitting jobs when *sent* is equals to the *GROUP_SIZE*.

Each consumer modelled is identified by an *id*. The value of *id* is used as the index to retrieve values from some data structures such as *cs* and *cp*, which are arrays of Booleans that store the current state for scheduler and the current partition of the consumer, respectively. These data structures are updated by the Partition template presented later which represents the election and recovery processes. If *cs[id]* is set to false, it means that the consumer is not associated with the scheduler; if *cp[id]* is false, it indicates that the consumer is in another partition.

Consumers also maintain a queue of the pending jobs (those submitted jobs that have not been completed yet). This array is accessed/modified by some of the procedures in the template:

- *addPending()*: inserts a job into the pending queue;
- *removePendingJob()*: removes a job from pending queue;
- *noPendingJobs()*: checks if the pending queue is empty or not.

7.2.2 Provider Template

Figure 83 presents the states and transitions for providers.

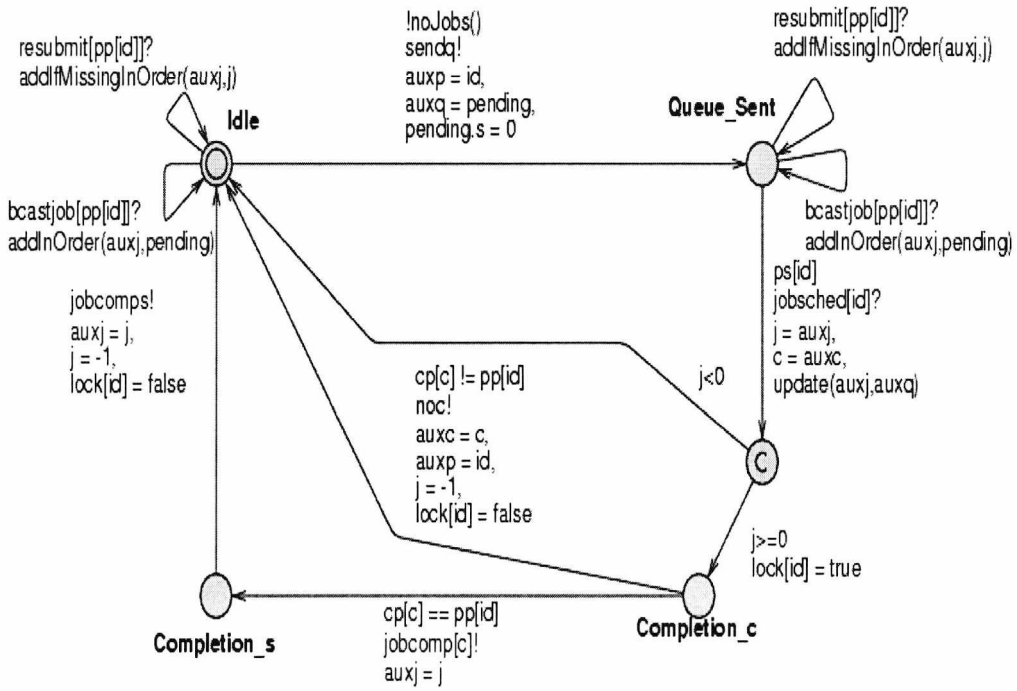

Figure 83 - Simple partition scenario: model of providers' behaviour.

In the *Idle* state, providers listen for messages broadcast by consumers (represented by the channels *bcastjobs* and *resubmit*). Once those messages are received, providers may add the jobs into a local queue for pending jobs. The procedures in the template that access/modify this queue are:

- *addInOrder()*: adds the jobs in the queue in the order of their global ID;
- *addIfMissingInOrder()*: similar to *addInOrder()*, but only adds to the queue if the job is not already in the queue;
- *noJobs()*: checks if the queue is empty or not;
- *update()*: updates the local queue based on the queue received from the scheduler.

If the local queue is not empty, a provider may send a queue fragment to the scheduler (represented by the channel *senq*) and move to the state *Queue_Sent*, where it waits for the response from the scheduler. While it waits, the provider can still receive the messages broadcast by consumers.

Providers receive the response from the scheduler through the *jobsched* channel, update their own local queue, and, if a job is scheduled ($j>=0$, where j is the selected job), providers go to the *Completion_c* state, where they can send the job completion message to the consumer that sent the selected job using the *jobcomp* channel.

It may happen that the consumer (whose id is represented by c) is not in the same partition ($cp[c] != pp[id]$, where *id* is the identifier of the provider, and *pp* is an array that indicates the partition of the provider) when the provider attempts to submit the completion message. In this case, providers inform the scheduler that consumer c is not available by using the *noc* channel, which will cause the scheduler to remove all the messages from c. Providers then return to the *Idle* state.

If consumer c is in the same partition ($cp[c] == pp[id]$), providers may go into the *Completion_s* state, where they can send the job completion message to the scheduler using the *jobcomps* channel.

When the job completion message is sent to the scheduler, providers return to the *Idle* state.

When providers receive the response from the scheduler, if no jobs have been scheduled ($j < 0$), providers return to the *Idle* state.

7.2.3 Scheduler Template

Figure 84 shows the states and transitions that represent the behaviour of the scheduler.

Similar to the consumers and providers, the scheduler also has a queue of pending jobs. Schedulers also maintain a queue of the jobs that have been allocated to a provider. These data structures can be accessed/modified by the following procedures:

- *add()*: inserts a job into the queue;
- *addResubmitted()*: adds a job, in order of global ID, to a queue, if it is missing;
- *deallocate()*: removes the specified job from the *allocated()* queue;
- *schedule_job()*: selects the next job to be allocated, removes it from the pending queue and then adds it to the allocated queue;
- *removeCons()*: removes all the jobs from a particular consumer from the pending queue.

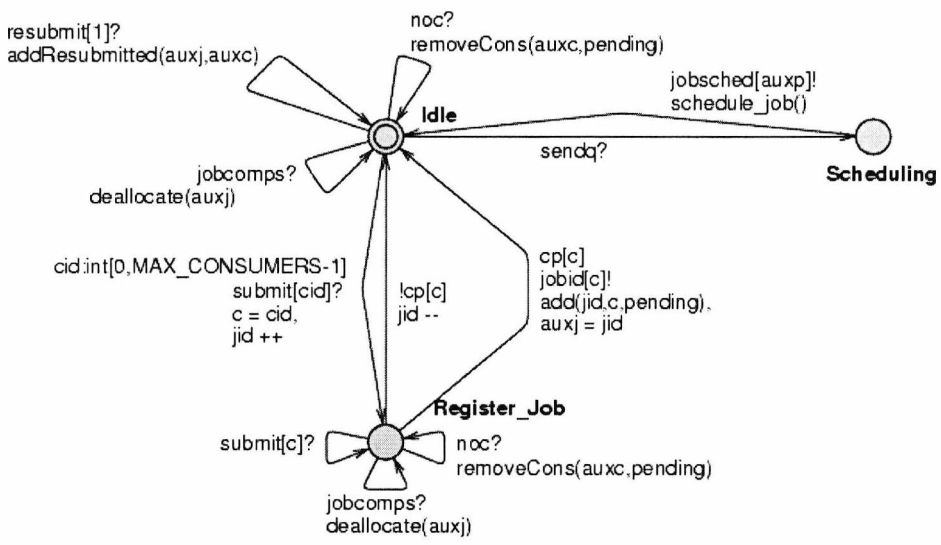

Figure 84 - Simple partition scenario: model of scheduler's behaviour.

From the *Idle* state, the scheduler can receive messages from consumers (*submit* and *resubmit* channels) and from providers (*sendq*, *jobscomps* and *noc* channels).

When the scheduler synchronises with the *submit* channel, it goes to the *Register_Job* state, where it can listen to *jobcomps* and *noc* channels. It may happen

that the scheduler takes too long to reply to the consumer, which will attempt to submit the job again; for this reason, the scheduler can still listen to the *submit* channel when it is in the *Register_Job* state, but only to the messages from the same consumer (*c*) that attempted it before.

From the *Register_Job* state, the scheduler goes back to *Idle* after adding the job to its own pending queue, sending the job's global ID to the consumer *c* (using the *jobid* channel) if the consumer is in the same partition. If the consumer is in a different partition (condition *!cp[c]*), the scheduler returns to the *Idle* state without adding the job to its pending queue, representing a communication timeout with the consumer.

When the scheduler receives the queue fragment from a provider (by synchronising on the *sendq* channel), it goes into the *Scheduling* state. The scheduler then returns to the *Idle* state after sending the selected job (if any) and the updated queue fragment to the provider (via the *jobsched* channel).

7.2.4 Partition Template

The template in Figure 85 shows the state machine that controls the modelled network partition case.

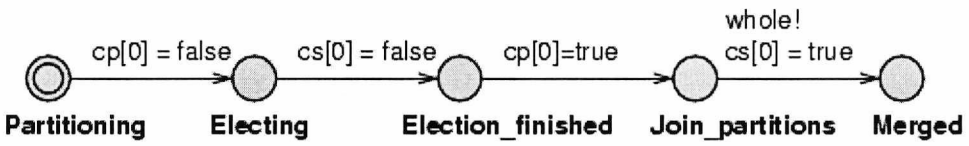

Figure 85 - Simple partition scenario: partition template.

In this template, we start by establishing that a particular consumer is separated from the main partition (*cp[0] = false*) with the occurrence of the transition from the *Partitioning* to the *Electing* state.

Once the partition has happened, the isolated consumer attempts to elect a scheduler. However, assuming that the device running the consumer is not scheduler capable, no scheduler will be elected. The transition from the *Election* to the *Election_finished* state represents the end of the failed election, where the consumer ends up with no scheduler associated with it (*cs[0] = false*).

The next transition (from the *Election_finished* to the *Join_partitions* state) returns the consumer to the main network, indicating the end of the partition.

The last transition (from *Join_partitions* to *Merged*) indicates that the consumer registers again with the grid, being associated to a scheduler.

7.2.5 Deadlock Verification

The model to be verified was set with 2 consumers, 2 providers and 1 scheduler, where one of the consumers becomes isolated in another partition.

We have defined that each consumer sends 2 jobs at most (*GROUP_SIZE = 2*). Such a low value was chosen with the assumption that it is enough to explore all the possibilities for the modelled scenario, without increasing the state space to a point where the model-checker runs out of computational resources to verify the model.

We have verified the model by asking the model-checker if there is a deadlock where there is a consumer that has not submitted the 2 jobs or have any job to be completed. The UPPAAL query that represents this property is:

```
E<> deadlock && exists (i:int[0,MAX_CONSUMERS-1])
(Consumer(i).sent < GROUP_SIZE ||
Consumer(i).pending.s>0)
```

The model-checker did not find a deadlock, which proves that our system continues to work when this case of network partition occurs.

7.3 Scenario 2: Complex Partition

In this scenario, we model a network partition case where two sub-networks coexist with a grid system working in each one of them, which means that the partition should last long enough for the devices to identify it and elect a new scheduler among them. This way we can verify the mechanisms for identifying the merge of the partitions, deciding which of the schedulers should continue to be active, and the synchronization between them.

In order for the election process to finish successfully (with a scheduler elected), we need a device that is capable of running the scheduler. Assuming that consumers are defined as non scheduler-capable, we have defined two partitions with a consumer and a provider each, where the devices running the providers can each be elected as the scheduler for its own partition.

Since, in this scenario, we are interested in verifying the system when there is more than one scheduler, we did not model the election process, and simply assume that it always happen before the partition merges.

The templates described in section 7.2 have been extended with more states and transitions, and the results are described below.

7.3.1 Partition Template

The template presented in Figure 86 describes all the steps for the modelled network partition.

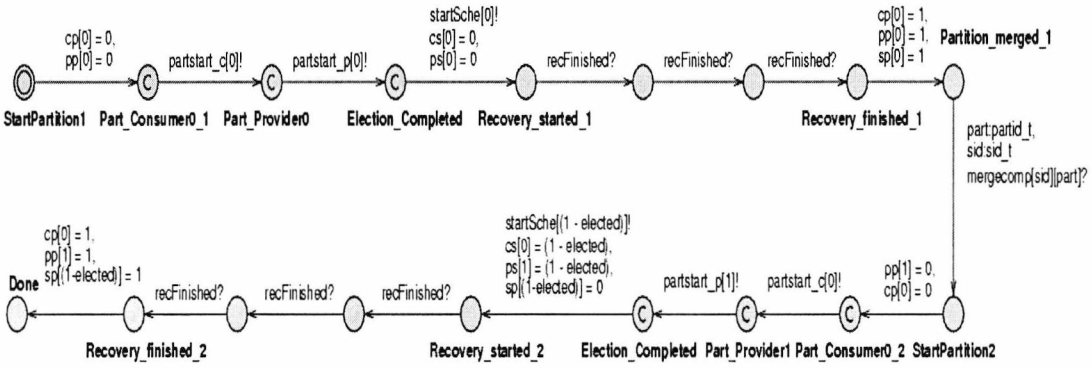

Figure 86 - Complex partition scenario: partition template.

Before the start of the partition, we assume that there are 2 consumers (*c0* and *c1*), 2 providers (*p0* and *p1*) and 1 scheduler (*s1*) active in the same network. We can also assume the case where *s1* runs in the same device as *p1*.

The transition from state *StartPartition1* to state *Part_Consumer0_1* represents the partition of the network into 2 distinct sub-networks (simply identified in the model as 0 and 1). In this transition, we also specify that *c0* and *p0* compose the partition 0 (*cp[0]=0* and *pp[0]=0*, respectively), while the other componentss (*c1*, *p1* and *s1*) compose the partition 1, which continues to work as if no partition has happened, since it has a scheduler. Note that, in the simple partition scenario we had *cp* as an array of Boolean values to indicate whether the device was in the main partition or not. However, in this scenario *cp* represents an array of integer values, indicating which partition the device belongs to.

The channels *partstart_c* and *partstart_p* correspond to control messages that inform consumers and providers, respectively, that the partition happened. These control channels are used to simplify the model and skip the election process, which was not modelled. The order of occurrence of these channels does not matter, and the

use of a committed location was meant to create some determinism and reduce the state space, allowing us to verify the model using the resources we have available.

We assume that the device running *p0* is elected to run the scheduler for the partition 0. This scheduler (*s0*) is started after synchronising with the *startSche* channel, in the transition from *Election_Completed* to *Recovery_started_1*, which also sets current scheduler for *c0* and *p0* (*cs[0]=0* and *ps[0]=0*, respectively).

According to the specification of our grid protocol in Chapter 4, the scheduler is only fully functional after the recovery process. For this reason, we have modelled that the partitions only merge after the recovery process, which is represented by the 3 consecutive synchronisations with the *recFinished* control channel; once for each component in the partition: *c0*, *p0* and *s0*.

The transaction from *Recovery_finished_1* to *Partition_merged_1* represents the merge of the partitions, with all devices becoming part of the partition 1.

The partition state machine can only move to the next state (*StartPartition2*) after synchronising with the *mergecomp* channel, which can only happen after one of the providers identifies that the partitions have been merged, and the schedulers have been synchronised, with one of them becoming inactive.

The transitions from *StartPartition2* to *Done* are similar to the ones from *StartPartition1* to *Partition_merged_1*, and represent another partition and merge process. The difference is that in the latter we have *c0* and *p0* forming partition 0, while in the former we have *c0* and *p1* composing that partition. This way we can verify cases where consecutive partitions may happen for different devices.

7.3.2 Consumer Template

The consumer template in Figure 87 extends the one presented in Figure 82 by having extra states and transitions to handle the recovery of the scheduler after its election, which does not happen in the simple partition scenario.

One of the new transitions is that which synchronises with the *partstart_c* channel, which indicates that the consumer is now part of another network partition, and makes the consumer go to the *Wait* state, where it waits for a message from the new scheduler via the *electcomp* channel. This message indicates that the election has been completed and that the consumer can start the recovery process (move to the *StartRecovery* state).

During the recovery process, consumers send the last global ID that they know (represented by *jsc* in the template). We modelled this using 2 channels: *nolst*, which is used if the consumer has not submitted any job so far (*jsc == 0*); and *recovered*, which is used to send to the scheduler the last job id known by the consumer, in the case where it has sent some job (*jsc > 0*).

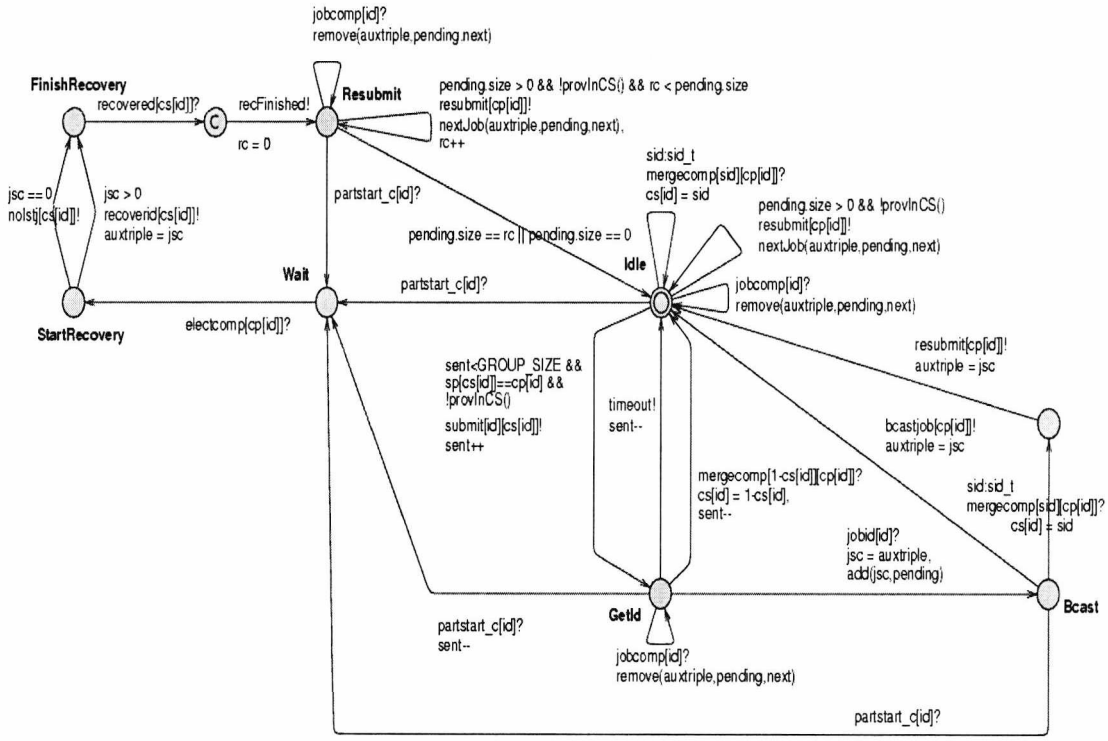

Figure 87 - Complex partition scenario: model of consumers' behaviour.

For each job, we needed to keep three pieces of information: the global ID, the consumer that owns the job and the scheduler that issued the ID. In the previous scenario (Simple Partition), we have used separate data structures (integer arrays) to represent these data, which requires more memory during the verification process. For this scenario, we have a single integer (*jsc*) composed with the job ID (*j*), the scheduler ID (s) and the consumer ID (*c*):

$$jsc = s \times 100 + c \times 10 + j$$

The first algorism of *jsc* represents the scheduler's ID, the second represents the consumer, and the third, the job ID. This composition works for a small number of jobs, which is the case in this model, since the increase of the number of jobs generates more states and requires more computational resources to verify it. The

default value for *jsc* is 0, which also indicates that no ID has been assigned to the job. To decode *jsc*, we use:

$$j = jsc \bmod 10$$
$$s = jsc/100$$
$$c = (jsc/10) \bmod 10$$

Once the synchronisation with any of these channels happens, consumers stay in *FinishRecovery* state, waiting for the message from the scheduler that informs it of the end of the recovery process, which is done via the *recovered* channel. After synchronising with this channel, consumers use the control channel *recFinish* to allow the partition machine to continue its own processes.

Another synchronisation that was added for this scenario is the one with the *mergecomp* channel, which is used to inform the winning scheduler after the merge of the partitions. Consumers only synchronise with this channel if the synchronisation request comes from a scheduler different from the one they currently know. When this happens, consumers update the information about the current scheduler.

7.3.3 Provider Template

The template in Figure 88 presents the extended behaviour of providers, so that they can handle the recovery of the scheduler and the merge of the partitions.

Whenever providers synchronise with the *partstart_p* channel, they go to the *WaitElection* state. Once in this state, providers start the recovery process after receiving the election completed message from the scheduler (*electcomp* channel), and then move to the *StartRecovery* state.

As part of the recovery process, providers send their queue of pending jobs and the queue of jobs that they have processed. In our model, this is done using two channels: *recoverq* and *recoverq_*, respectively. Once the queues are submitted, providers wait in the *FinishRecovery* state, until the scheduler sends a message informing them of the end of the recovery process (using the *recovered* channel). When this happens, providers use the control channel *recFinish* to allow the partition machine to continue its own processes and then go back to the *Idle* state.

As specified in Chapter 4, a provider can identify the merge of partitions when it receives a broadcast message with the description of a job with the global ID issued by a scheduler that is different from the one known by the provider. In the provider template, the *bcastjob* channel represents this message.

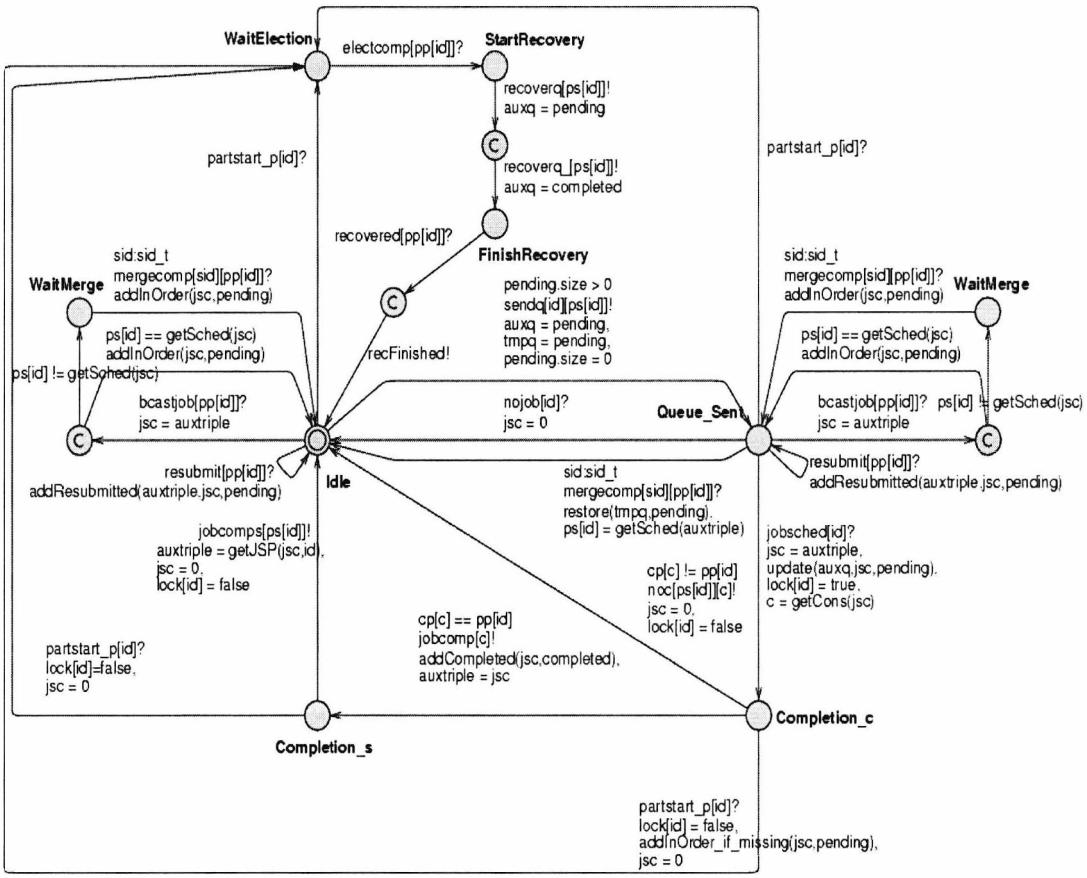

Figure 88 - Complex partition scenario: model of providers' behaviour.

If the global ID of the broadcast job is issued by the known scheduler (*ps[id] == getSched(jsc)*), providers add the job to their own pending queue and return to the *Idle* state. Otherwise (if *ps[id] != getSched(jsc)*), providers go to the *WaitMerge* state and wait for the merge of the schedulers to finish; at the same time, a sub process is started to send a message to both schedulers to make them aware of each other. Figure 89 represents the state machine (*Pmerger* template) for this sub process.

Pmerger starts at the *Idle* state and synchronises with the *bcastjob* channel, moving to the *StartMerge* state if the job's global id was issued by an unknown

scheduler. Once in that state, Pmerger sends a MERGE message to both schedulers using the *merge* channel, and then moves to the *Merging* state.

When a MERGE_COMPLETE message arrives (via the *mergecomp* channel), Pmerger updates the provider's scheduler (*ps[i] = sid*, where *sid* is the ID of the scheduler who sent the message), and then returns to the *Idle* state.

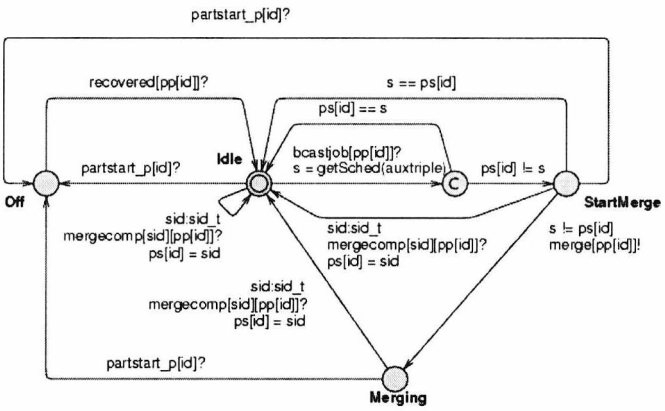

Figure 89 - Complex partition scenario: Pmerger template.

If, in any state, another partition happens (synchronising with the *partstart_p* channel), Pmerger goes to the *OFF* state, and returns to *Idle* after the elected scheduler completes the recovery process and the synchronisation of the *recovered* channel happens.

7.3.4 Scheduler Template

The template in Figure 90 shows the behaviour of the scheduler in the complex partition scenario.

In this model, the elected scheduler is started by synchronizing with the partition state machine via the *startSched* channel, then moving to the *ElectionComplete* state.

The scheduler then announces its election to consumers and providers via the *electcomp* broadcast channel, and then moving to the *StartRecovery* state, where it can wait for the last known job ID from consumers and the queues from providers. After receiving all the recovery data, the scheduler goes to the *FinishRecovery* state, where it can inform the other components that the recovery process is complete (via the *recovered* channel). The scheduler then uses the *recFinish* channel to allow the partition machine to continue its own processes.

Since we assume that the partition lasts long enough for the recovery process to finish, we do not have to model the timeouts for the election or recovery.

When schedulers receive a MERGE message from a provider, they go to the *Merging* state, where one of them is non-deterministically selected (this is to simplify the model by not computing a utility value for each scheduler).

The selection of the winning scheduler is done with a help of the state machine presented in Figure 91.

Since the *Merging* state is a committed location, UPPAAL non-deterministically chooses one of the schedulers to synchronize with the *pickme* channel first, forcing the other scheduler to send the SYNC message (using the *sync* channel) carrying the queues with the pending and allocated jobs and then becomes unavailable (*OFF* state).

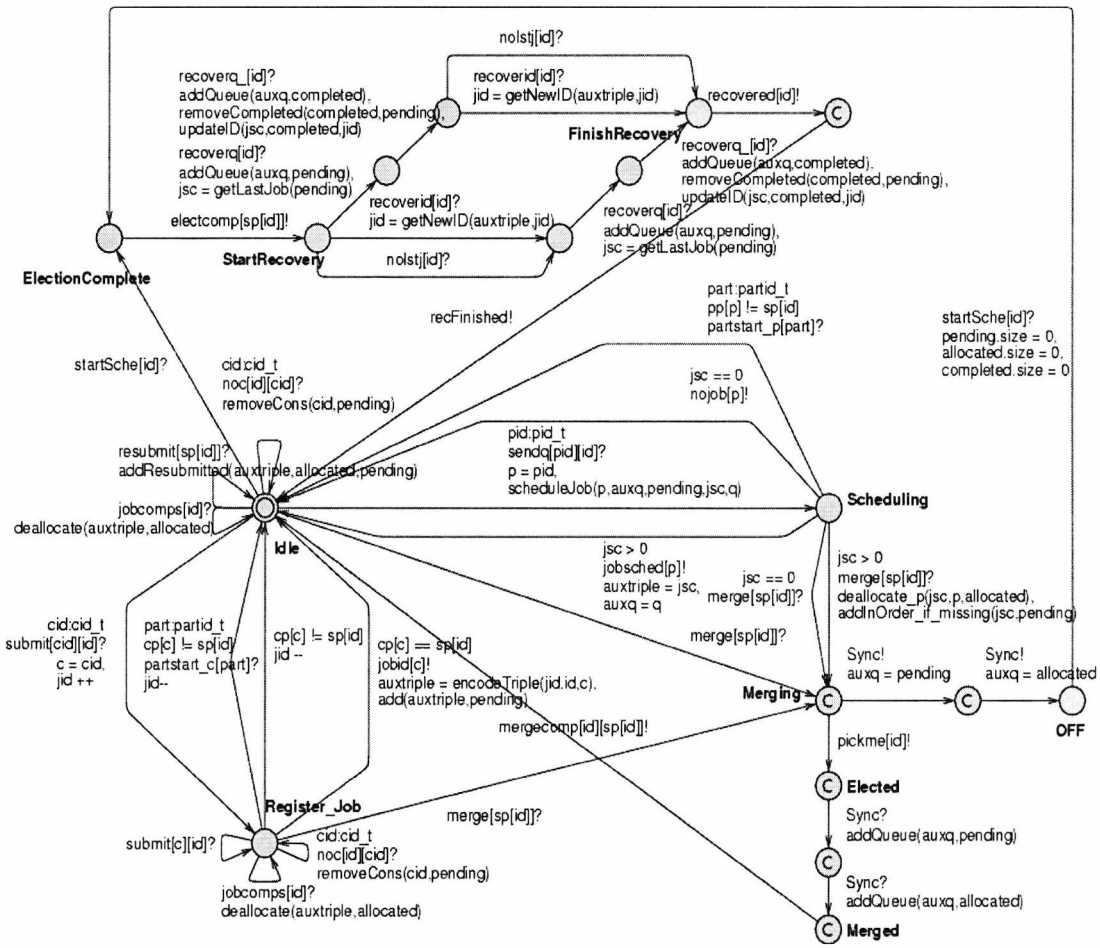

Figure 90 - Complex partition scenario: model of scheduler's behaviour.

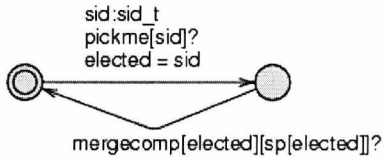

Figure 91 - Template for random selection of scheduler.

The winning scheduler receives the SYNC message with the queues and then updates its own queues using the *addQueue(A, B)* procedure (adds the elements from queue A into B, if they are not yet in B).

When the synchronisation is over, the winning scheduler (currently in the *Merged* state) notifies the other components using the *mergecomp* channel, and then returns to the *Idle* state.

The synchronisations with the *partstart_c* and *partstart_p* were added to the scheduler template to model network timeouts after the partition of the network.

7.3.5 Deadlock Verification

We tried to verify the model using 2 consumers (with *GROUP_SIZE = 2*), 2 providers and 2 schedulers. Unfortunately, the state space for this scenario is very big, and UPPAAL ran out of computational resources. We have used a machine running Linux and with 6 GB of RAM memory, but UPPAAL can only use 3 GB of RAM and it was not able to complete the verification.

To solve this problem, we have created an abstract model for this scenario, which has a smaller state space than the concrete version and it does not use data structures for storing the information about jobs, thus consuming less memory.

Figure 92 shows the abstract provider template. Note that, for example, instead of invoking the *addResubmitted()* procedure when synchronising with the *resubmit* channel (see Figure 88), it simply sets a Boolean variable to indicate that there is something in the queue.

In the abstract model, there is only the idea that a job is scheduled or not, without being specific about what job. When providers receive a job to be processed, it non-deterministically assigns a consumer to the job, since no information about the jobs is stored.

APPENDIX E presents all the abstract templates.

This abstraction brings many possibilities that would not happen in the concrete model; however, all the behaviours of the concrete model are preserved. Therefore, assuming that all the possible deadlocks in the concrete model are also preserved in the abstract one (see proof in the next section), we can assert that if no deadlock is found with the abstract model, there is no deadlock in the concrete model either.

We checked the abstract model for deadlocks using the following UPPAAL query:

```
E<> (deadlock && exists (c:cid_t)
Consumer(c).completed < GROUP_SIZE)
```

This query asks UPPAAL if there is any deadlock with any job pending completion. After running it, the model checker verified that there is no deadlock, showing that the proposed system also works in this complex partition scenario.

We also attempted to verify the abstract model with GROUP_SIZE = 3, but UPPAAL ran out of memory and could not complete the verification.

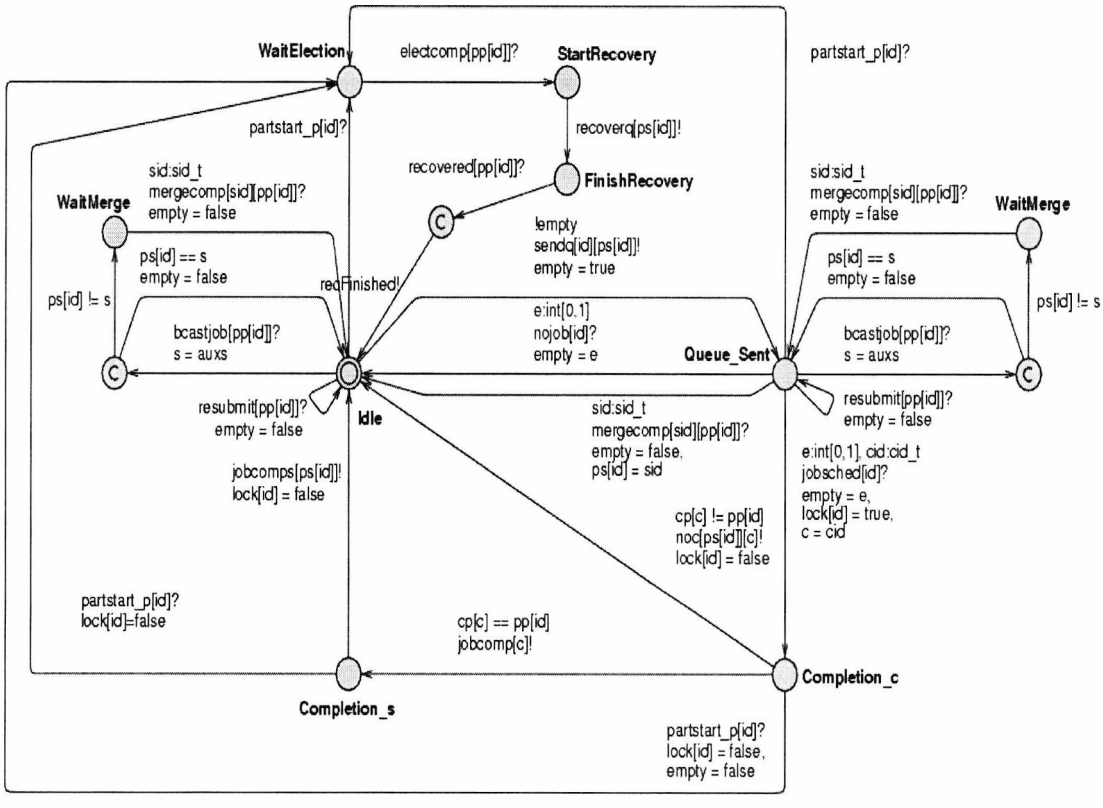

Figure 92 - Abstract Provider Template.

7.4 A proof of deadlock preservation with abstract models

Let C and A denote two UPPAAL networks for a given job scheduling protocol over grids. In particular, C, the concrete model, is defined by the UPPAAL network `doublepart_nano.xml`. A, the abstract model, is defined by the network `doublepart_abstract.xml`.

We omit here the formal semantics of UPPAAL timed automata networks (see e.g. [111] and [112]). For the purpose of this proof, it suffices to assume that the behaviour of the networks, C and A, are given by transition systems [113].[6] A transition system is a tuple (S, s_0, T), where S is the set of reachable states, $s_0 \in S$ is the initial state, and $T \subseteq S \times S$ is the transition relation. A state in the transition system is a tuple $s = (\bar{l}, v)$, where $\bar{l}$ is a location vector and v is a valuation. A location vector is a vector $\bar{l} = \langle l_0, \ldots, l_n \rangle$, where n is the number of automata in the network and l_i, $i : 1..n$, is a location in the i-th automata. We will use $l \in \bar{l}$ to denote that location l is in the location vector $\bar{l}$. A valuation v maps each variable in the network to a value in its type. We will use $v(x)$ to denote the value of variable x in v. We will use $v \models \phi$ ($v \not\models \phi$) to denote that the predicate ϕ is satisfiable (respectively unsatisfiable) in valuation v. Transitions in T denote state changes due to the execution of an internal transition of a component in the network or the synchronization among different components of the network. We will use $s \rightarrow s'$ to denote a transition from state s to state s' (where s and s' are not necessarily different).

Let $C = ((C_0, \ldots, C_7), V_c, Ch_c)$ and $A = ((A_0, \ldots, A_6), V_a, Ch_a)$, where C_0 = Consumer, C_1 = Provider, C_2 = Scheduler, C_3 = Partition, C_4 = Pmerger, C_5 = ElectSched, C_6 = Initializer, C_7 = Exception, and $A_i = C_i$ for $i : 0..6$ (the template Exception is not included in the abstract model). Let $\mathcal{T}_C = (S_C, s_{c,0}, T_C)$ and $\mathcal{T}_A = (S_A, s_{a,0}, T_A)$ denote the transition systems of C and A, respectively. Let $\sim_s \in S_C \times S_A$ be an equivalence behavioural relation between the states of C and A. The relation $\sim_s$ is defined so that, for any concrete state $s_c = (\bar{l}_c, v_c) \in S_C$ and any abstract state $s_a = (\bar{l}_a, v_a) \in S_A$, $s_c \sim_s s_a$ holds if and only if the following conditions hold:

1. The relation between locations in the concrete model and locations in the abstract model is defined by the following predicates:

   $\texttt{Provider.c3} \in \bar{l}_c \Rightarrow \texttt{Provider.ElectionCompleted} \in \bar{l}_a$

[6] There is no need to resort to *timed* transition systems, since our models do not have clocks.

Scheduler.u11 $\in \bar{l}_c \Rightarrow$ Scheduler.ElectionCompleted $\in \bar{l}_a$
Scheduler.u12 $\in \bar{l}_c \Rightarrow$ Scheduler.u1 $\in \bar{l}_a$
Scheduler.u21 $\in \bar{l}_c \vee$ Scheduler.u22 $\in \bar{l}_c \Rightarrow$ Scheduler.u2 $\in \bar{l}_a$
Scheduler.c1 $\in \bar{l}_c \Rightarrow$ Scheduler.Bid $\in \bar{l}_a$
Scheduler.c2 $\in \bar{l}_c \Rightarrow$ Scheduler.Elected $\in \bar{l}_a$
$l \in \bar{l}_c \Rightarrow l \in \bar{l}_a$ for all other location l

2. The relation between variables in the concrete model and variables in the abstract model is defined by the following predicates:

Consumer.GetId $\in \bar{l}_c \Rightarrow v_c$(Consumer.sent) = v_a(Consumer.sent) + 1
Consumer.GetId $\notin \bar{l}_c \Rightarrow v_c$(Consumer.sent) = v_a(Consumer.sent)
v_c(Consumer.sent)>0 $\Rightarrow v_a$(Consumer.sent) = v_c(Consumer.sent)
v_c(Consumer.completed) = v_a(Consumer.completed)
v_c(Consumer.pending.size) = v_a(Consumer.sent) – v_a(Consumer.completed)
v_c(Consumer.jsc) / 100 = v_a(Consumer.s)
v_c(Consumer.jsc) > 0 $\Leftrightarrow v_a$(Consumer.got_job_id)
v_c(Consumer.id) = v_a(Consumer.id)
v_c(Consumer.rc) = v_a(Consumer.rc)
v_c(Provider.pending.size) = 0 $\Leftrightarrow v_a$(Provider.empty)
Provider.procesing $\in \bar{l}_c \Rightarrow (v_c$(Provider.jsc) / 10)$_{\equiv 10} = v_a$(Provider.c)
Provider.c1 $\in \bar{l}_c \vee$ Provider.c2 $\in \bar{l}_c \Rightarrow v_c$(Provider.jsc) / 100 = v_a(Provider.s)
v_c(Provider.id) = v_a(Provider.id)
v_c(Scheduler.id) = v_a(Scheduler.id)
v_c(Scheduler.c) = v_a(Scheduler.c)
v_c(Scheduler.p) = v_a(Scheduler.p)
$\forall$ *var* $\in$ {cp,pp,cs,ps,sp,lock,elected}. v_c(*var*) = v_a(*var*)

Let Ch_c and Ch_a denote the sets of channels in C and A, respectively. Let $\sim ch$: $Ch_c \times Ch_a$ so that recoverq_ $\sim_{ch}$ recoverq and $ch \sim_{ch} ch$ for any $ch \neq$ recoverq_.

THEOREM 1.1. *The following conditions hold*:

1. *The initial states in the concrete and the abstract model are equivalent.*

$$s_{c,0} \sim_s s_{a,0}$$

2. *The abstract model simulates the concrete model.*

$$\forall s_c, s_a, s'_c.\ s_c \sim_s s_a \wedge s_c \rightarrow s'_c \in T_C \Rightarrow \exists s'_a.\ s'_c \sim_s s'_a \wedge s_a = s'_a \vee s_a \rightarrow s'_a \in T_A$$

3. *A state in the concrete model is a deadlock if and only if the equivalent state in the abstract model is a deadlock.*

$$\forall s_c, s_a.\ (s_c \sim_s s_a \wedge \nexists s'_c.\ s_c \rightarrow s'_c \in T_C) \Rightarrow \nexists s'_a.\ s_a \rightarrow s'_a \in T_A$$

Proof. The first condition holds trivially by definition of $s_{c,0}$, $s_{a,0}$ and $\sim_s$.

We will omit a complete proof of the second condition, which is a tedious and rather standard proof of simulation (as commonly found in process calculi

textbooks). Such a proof can be simply obtained by cases, and follows by definition of $\sim_s$ and $\sim_{ch}$. Instead, by way of example, we sketch here the proof for the following case:

1. $s_c \sim_s s_a$;
2. {Consumer.Resubmit, Scheduler.Idle} $\subset \bar{l}_c$; and
3. s'_c is the next-state that results from the synchronization between Consumer and Scheduler on channel resubmit.

Let $s_c = (\bar{l}_c, v_c)$, $s_c = (\bar{l}'_c; v'_c)$ and $s_c = (\bar{l}_a, v_a)$. We want to prove that there exists $s'_a = (\bar{l}'_a, v'_a)$ so that $s'_c \sim_s s'_a$ and either $s_a = s'_a$ or there exist a transition $s_a \rightarrow s'_a \in T_A$.

Note that resubmission in the concrete model implies that (1) the corresponding transition is enabled, and (2) the channels match. Formally,

1. $v_c \models$ Consumer.pending.size $>$ 0 $\wedge$ $\neg$provInCS() $\wedge$ Consumer.rc $<$ Consumer.pending.size
2. $v_c \models$ cp[Consumer.id] = sp[Scheduler.id]

Once synchronisation occurs, we have that v'_c(Consumer.rc) = v_c(Consumer.rc) + 1 and all other relevant variables (that is, those concrete variables that are related to abstract variables) remain unchanged.

Now, by definition of $\sim_s$, and given the hypothesis $s_c \sim_s s_a$, it is the case that

1. {Consumer.Resubmit, Scheduler.Idle} $\subset \bar{l}_a$,
2. $v_a \models$ Consumer.sent – Consumer.completed $>$ 0 $\wedge$ $\neg$provInCS() $\wedge$ Consumer.rc $<$ Consumer.sent – Consumer.completed, and
3. $v_a \models$ cp[Consumer.id] = sp[Scheduler.id].

Then, at s_a, synchronisation over channel resubmit can also occur between Consumer and Scheduler in the abstract model, and this results in a next-state s'_a so that v'_a (Consumer.rc) = v_a(Consumer.rc) + 1 and all other relevant variables remain unchanged. Hence, by definition of $\sim_s$, $s'_c \sim_s s'_a$ (qed).

The third condition can be proved by contradiction, following reasoning similar to the one that proves the second condition.

COROLLARY 1.2. For any reachable deadlock state s_c in the concrete model, there exists an equivalent reachable deadlock state s_a in the abstract model (that is, $s_c \sim_s s_a$).

THEOREM 1.3. *Let s_c be a state in the concrete model, and $s_a \sim_s s_c$ be an equivalent state in the abstract model ($s_a \sim_s s_c$). If s_c satisfies* Consumer.sent < *GROUP_SIZE or it satisfies* Consumer.pending.s > *0, then sa satisfies* Consumer.completed < *GROUP_SIZE.*

Proof. Follows trivially from the definition of $\sim_s$, and the invariant Consumer.sent ≤ *GROUP SIZE*, which holds both in the concrete and in the abstract model.

7.5 Discussion

During the verification of the scenario 1, we were able to spot a problem with our protocol that was causing UPPAAL to identify a possible deadlock for the system.

In the initial modelled version of our system (M0), the job resubmission message was sent as a TCP message from consumers to the scheduler. After receiving such a message, the scheduler would then contact the provider to which the job was assigned (if any).

After submitting a job to the scheduler, consumers broadcast the description of the jobs, so that providers can find out about them and process the matchmaking. Since we do not assume the use of reliable broadcast, it may happen that active providers do not receive some broadcast messages from consumers.

The case that caused the deadlock with M0 was when all providers are active during the whole process and none of them received the broadcast message with the description of a certain job. In this case, only the consumer and the scheduler are aware of that job, thus the job cannot be allocated to any of the active providers.

According to M0, when the consumer gets a timeout for the job, it sends a reminder to the scheduler, which only takes further action if the job is allocated. Since the job has not been allocated, none of the active providers would find out about the job, and it would never be processed in this scenario, unless the consumer has the policy of executing the job itself after a certain number of attempts.

The solution for this problem was to convert the job resubmission message into a broadcast message, giving the possibility to providers to receive it, and add it to their own queues, so the job can be allocated to one of them.

This kind of problem could not be identified solely using our simulations, since all the broadcast messages in the simulation are delivered.

Chapter 8. Conclusions and Future Work

8 Conclusions and Future Work

This chapter presents the conclusions about the main contribution from this thesis, and it presents possible work than can be carried out in the future.

8.1 Conclusions

In this thesis, we have described the home environment as a possible scenario where the concepts of grid computing can be applied. We have introduced a TV recommender running SVD as a possible application for the system. Other examples of applications could be a security system that analyses video images or health systems that require more processing power to constantly monitoring data from a patient in the house. The home grid can also serve as a resource for conventional grid applications such as SETI@Home and FightAIDS@Home.

Here we considered the fact that the grid scheduler should be able to run in a limited device, and we showed that the majority of the scheduling and fault tolerance techniques applied to conventional grid systems are not suitable for the home environment. Based on this, we proposed a simple P2P based scheduling protocol that allows limited devices to play the role as the scheduler. This scheduling protocol represents the main contribution of this thesis, and it can be applied to other highly dynamic and heterogeneous environments.

Our protocol specifies how devices are registered with the grid and how resources are found using broadcasting of jobs descriptions that carry the minimal requirements for the jobs execution. Our scheduling and allocation mechanism was designed to reduce loss of jobs and improve the throughput in a constrained environment, also allowing providers to recover jobs sent to the system while they were unavailable.

Our solution also takes into consideration the fact that some consumer devices may be running on battery power, and presents low communication overhead.

We have also described how our protocol handles faults, and we proposed a distributed election mechanism that is used to choose a new scheduler when the current one fails. We have also explained how the state of the scheduler is recovered,

while avoiding extra communication and processing that would be needed for the resubmission of all jobs.

Another contribution from this thesis is the simulator, which, unlike other grid simulators, provides support for broadcasting and a simple version of the TCP protocol for the communication. Another difference is that, instead of *makespan*, it supports throughput as one of the metrics, which is ideal for evaluating task independent schedulers.

We have simulated our solution and compared it with OLB using parameters that give us extreme situations in order to evaluate the performance and scalability of both systems, and to make sure that there are no hidden bugs in our design. The results have shown that our solution performs similar to or better than that provided by OLB, plus the fact that our solution supports fault tolerance and scales with the increase of the number of providers.

To complete this work, we have also formally modelled (using UPPAAL) part of the protocol and verified two cases of network partition, showing that our solution can tolerate those two scenarios of network failures. Other grid systems usually assume the existence partition-free network topology (which is not the case of common home networks) and do not consider such type of failure.

An additional contribution from this thesis is the resource description model presented in Appendix B, which can describe the variety of devices that can be found in a home and be used to build a complete home grid system in that environment.

8.2 Overall Contributions

As mentioned in the previous section, the main contribution of this thesis are the definition of the requirements for a home grid and the fault-tolerant distributed scheduling system that can run in a limited device and that does not require much processing from the consumers for the scheduling and the fault-tolerant mechanisms.

The other major contribution of this thesis is the simulator that supports the throughput as the evaluation metric. The simulation core developed at the University of Kent only provides the clock that controls the execution of events based on a queue and the interfaces for the development of the simulation objects/components.

We have extended the simulation core to be able to remove scheduled events. We also implemented all the grid components (consumer, scheduler, provider, etc.) for our proposed solution and for OLB, the components for the network simulation

(including the simplified TCP and the broadcasting using UDP explained in the Chapter 5) and the component that simulates the processor based on the Mips and MFlops.

8.3 Future Work

The focus of this thesis is on the scheduling mechanism for a home grid, but we have also presented an component-based architecture for the whole grid. Since we have only simulated the system, the immediate future work that comes to mind is the real implementation of it and its deployment on real devices.

The complete implementation of the system should start by the definition of interfaces for the software components specified in the section 3.5, which will allow different implementations of each component without affecting the functionality of the others. For example, some limited devices may require specific implementation for the communication component due to the variety of technologies available.

Even though we have not specified the security mechanisms for the system, we can still have a functional implementation of the system without the security aspects for testing purposes. Once the implementation is complete, it can be tested with simulated workload or with a real application such as a recommender system.

The following sections present other future work that can be developed.

8.3.1 Simulation parameters

Even though we have simulated a number of different configurations, we did not exhaust all the possibilities because of time constraints or because they were out of the scope of this thesis. For example, in our experiments, the providers had the same capabilities, and we assumed that they were all able to run all the submitted jobs; further simulations could involve different types of jobs and a more heterogeneous environment (we have mainly focused on the dynamicity of the environment and on the limitations of the scheduler device).

We have invested a considerable amount of time and effort in making sure that the simulations are correct and in trying to explore situation where our system could demonstrate a considerable gain in performance. For this reason, we have mostly used extreme scenarios that do not necessarily reflect the home environment. Future simulations should attempt to reflect a home environment better, e.g. the use of

fewer consumers, the addition of input and output data and their location, consumers becoming unavailable, etc.

8.3.2 More schedulers

One can develop and/or simulate other types of schedulers and compare the results with the ones from our solution. One possibility to consider is a scheduler that is fully distributed (i.e., without a central device to make the final decision), which could be an interesting choice for comparison against our proposed solution.

A further improvement to our scheduling solution could focus on the reduction of the processing power required for making the final scheduling decision after receiving the list of jobs from the providers. In our simulations, we have implemented the scheduler by using a hash map as the scheduler's local data structure, and a linked list to represent the list sent by the providers. Further optimization can still be made to the algorithms and data structures, reducing the CPU usage and scheduling time.

8.3.3 Security

It is necessary to define security mechanisms and policies to protect the information transferred in the home grid. Such security solutions should take into consideration the limitations of the environment.

The main aspects of security that must be addressed internally in the home grid system, i.e. not considering external access (e.g. to or from the Internet), are:

(i) Authentication: devices must be able to authenticate each other by themselves or via another device with better capacities in which they have trust. The second option is needed due to the device's limitations (processing, storage and power).

(ii) Access control: every device has control over its own resources; for this reason an access policy must be elaborated, specifying which resources can be used by a certain device and how they can be used.

(iii) Confidentiality: applications may require that some data must be protected against unauthorized entities. Because of this, the grid system must give support to such a requirement. Nevertheless, the limitations of the devices impose a challenge when attending to this requirement, since some

of them may not have enough capacity to run encryption algorithms with a safer key length. The grid system must then be able to adapt to such limitations [114].

(iv) Integrity: the data must arrive at the target device unchanged; otherwise applications will start giving wrong results.

(v) Availability: the central scheduler must be protected against *denial of service* attacks that make it become unavailable.

There are many researchers working on secure infrastructures for sensor networks, and perhaps some of the solutions for those environments can be applied to the home grid.

8.3.4 Model Checking

As mentioned earlier, we have not modelled the full behaviour of the system, mainly due to time constraints and limitations of the tools to handle the size of the state space that can be generated.

Parts of the system that have not been verified formally include the election and the recovery processes, which have only been simulated.

Other network partition cases can also be modelled, e.g. the partition happening while the election or the recovery is in progress. We believe that our protocol can handle these cases, but we did not formally verify them for the reasons listed in the beginning of this section.

8.3.5 Data Grid

Our solution considers the existence of a computational home grid where the locations of the input and output (I/O) data required by the jobs are specified with their description.

Considering the existence of many different storage places in the house, there is a possibility of designing a home data grid [104] to facilitate the location of data, and to use such a system to store and locate I/O data for the jobs in the computational grid.

References

1. Adomavicius, G., et al., *Incorporating Contextual Information in Recommender Systems Using a Multidimensional Approach.* ACM Transactions on Information Systems (TOIS), 2005. **23**(1): p. 103 - 145.
2. Adomavicius, G. and A. Tuzhilin, *Toward the Next Generation of Recommender Systems: A Survey of the State-of-the-Art and Possible Extensions.* IEEE Transactions on Knowledge and Data Engineering, 2005. **17**(6): p. 734-749.
3. Gutta, S., et al., *TV Content Recommender System*, in *17th National Conference on Artificial Intelligence and 12th Conference on Innovative Applications of Artificial Intelligence*. 2000, AAAI Press / The MIT Press: Austin, TX.
4. Konstan, J.A., et al. *Recommender Systems: A GroupLens Perspective*. in *Recommender Systems: Papers from the 1998 Workshop (AAAI Technical Report WS-98-08)*. 1998. Menlo, CA: AAAI Press.
5. Sarwar, B., et al. *Incremental SVD-Based Algorithms for Highly Scalable Recommender Systems*. in *Fifth International Conference on Computer and Information Technology (ICCIT 2002)*. 2002.
6. Sarwar, B.M., et al., *Application of Dimensionality Reduction in Recommender System -- A Case Study*, in *Web Mining for E-Commerce -- Challenges and Opportunities (WEBKDD'2000)*. 2000: Boston, MA, USA.
7. Kalman, D., *A Singularly Valuable Decomposition: The SVD of a Matrix.* The College Mathematics Journal, 1996. **27**(1): p. 2-23.
8. Oksa, G., M. Becka, and M. Vajtersic, *On a Parallel Implementation of the One-Sided Block Jacobi SVD Algorithm*. 2007: http://www2.cs.cas.cz/harrachov/slides/Oksa.pdf.
9. Agarwal, A., D.O. Norman, and A. Gupta, *Wireless Grids: approaches, architectures, and technical challenges*, in *MIT Sloan School of Management*. 2004. p. 9.
10. Ahuja, S.P. and J.R. Myers, *A Survey on Wireless Grid Computing.* The Journal of Supercomputing, 2006. **37**: p. 3-21.

11. Chu, D.C. and M. Humphrey. *Mobile OGSI.NET: Grid Computing on Mobile Devices*. in *5th IEEE/ACM International Workshop on Grid Computing*. 2004. Pittsburgh, PA.
12. Foster, I., C. Kesselman, and S. Tuecke, *The Anatomy of the Grid: Enabling Scalable Virtual Organizations.* The International Journal of High Performance Computing Applications, 2001. **15**(3): p. 200-222.
13. Lioupis, D., D. Psihogiou, and M. Stefanidakis, *Exporting Processing Power of Home Embedded Devices to Global Computing Applications*, in *12th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP'04)*. 2004, IEEE Computer Society: Los Alamitos, CA, USA. p. 274.
14. Shen, H., et al., *Grid-Based Multi-scale PCA Method for Face Recognition in the Large Face Database*, in *Advanced Web and Network Technologies, and Applications*. 2006, Springer Berlin / Heidelberg. p. 1033-1040.
15. Nicholl, P., A. Amira, and R. Perrott, *An Automated Grid-Enabled Face Recognition System using Hybrid Approaches*, in *Postgraduate Research Conference in Electronics, Photonics, Communications and Networks, and Computing Science, University of Lancaster*. 2005: Lancaster, UK.
16. Martínez, D., et al. *Wireless Sensor and Actuator Networks: Characterization and Case Study for Confined Spaces Healthcare Applications*. in *International Multiconference on Computer Science and Information Technology*. 2008: IEEE.
17. Stankovic, J.A., et al., *Wireless Sensor Networks for In-Home Healthcare: Potential and Challenges*, in *High Confidence Medical Device Software and Systems (HCMDSS) Workshop*. 2005: Philadelphia, PA, USA.
18. Nam, D.S., et al. *QoS-Constrained Resource Allocation for a Grid-Based Multiple Source Electrocardiogram Application*. in *International Conference on Computational science and its applications: ICCSA 2004*. 2004. Assisi, Italy: Springer-Verlag Berlin Heidelberg.
19. Zhang, H., H. Ma, and A. Ming. *Grid-Based Multi-scale PCA Method for Face Recognition in the Large Face Database*. in *APWeb Workshops 2006*. 2006: Springer-Verlag Berlin Heidelberg.

20. Grgic, M., K. Delac, and S. Grgic, *SCface - surveillance cameras face database.* Multimedia Tools and Applications Journal, 2011. **51**(3): p. 863-879.
21. Bischof, C.H., *Computing the Singular Value Decomposition on a Distributed System of Vector Processors*. 1987, Cornell University: Ithaca, NY, USA. p. 24.
22. ANCL. *Physio Grid.* 2008 [cited 2011 04/04/2011]; Available from: http://ancl.kaist.ac.kr/project/physio.html.
23. Jones, C.E., et al., *A Survey of Energy Efficient Network Protocols forWireless Networks.* Wireless Networks, 2001. **7**: p. 343–358.
24. Chandrakasan, A.P., S. Sheng, and R.W. Brodersen, *Low-Power CMOS Digital Design.* IEEE Journal of Solid-state Cirtcuits, 1992. **27**(4): p. 473-484.
25. Balani, R., *Energy Consumption Analysis for Bluetooth, WiFi and Cellular Networks*. 2007, University of California at Los Angeles: Los Angeles, California. p. 6.
26. Miettinen, A.P. and J.K. Nurminen. *Energy efficiency of mobile clients in cloud computing*. in *2nd USENIX Workshop on Hot Topics in Cloud Computing*. 2010.
27. Directgov. *Consumer Durables Consumer durables ownership increases*. 2010 14/01/2010 [cited 2010 10/03/2010]; Available from: http://www.statistics.gov.uk/cci/nugget.asp?id=868.
28. Khargarhia, B., *Game consoles grow up, go after adult demographic*. 2007, The Money Times: http://www.themoneytimes.com/articles/20070314/game_consoles_grow_up_go_after_adult_demographic-id-103096.html, last accessed on 07/10/2007.
29. Song, H., et al. *UPnP-Based Sensor Network Management Architecture*. in *Second International Conference on Mobile Computing and Ubiquitous Networking (ICMU 2005)*. 2005. Osaka, Japan.
30. Dobrescu, R., et al., *Embedding wireless sensors in UPnP services networks.* International Journal of Communications, 2007. **1**(2): p. 62-67.
31. Baker, M., R. Buyya, and D. Laforenza, *Grids and Grid technologies for wide-area distributed computing.* Software-Practice and Experience (SP&E), 2002. **32**: p. 1437-1466.

32. Foster, I. and C. Kesselman, eds. *The Grid: Blueprint for a Future Computing Infrastructure.* 1999, Morgan Kaufmann: San Francisco, CA.
33. Vraalsen, F., et al. *Performance Contracts: Predicting and Monitoring Grid Application Behavior.* in *2nd International Workshop on Grid Computing.* 2001. Denver, Colorado, USA: Spring-Verlag.
34. Deerwester, S., S.T. Dumais, and R. Harshman, *Indexing by Latent Semantic Analysis.* Journal of the American Society for Information Science, 1990. **46**(6): p. 391-407.
35. Foster, I. and C. Kesselman, *Globus: A Metacomputing Infrastructure Toolkit.* International Journal of Supercomputer Applications, 1997. **11**: p. 115-128.
36. Dong, F. and S.G. Akl, *Scheduling Algorithms for Grid Computing: State of the Art and Open Problems*, in *Technical Report No. 2006-504.* 2006, School of Computing, Queen's University: Kingston, Ontario, USA. p. 55.
37. Legion, *Legion - A Worldwide Virtual Computer: Legion Overview.* 2007: http://legion.virginia.edu/overview.html, last accessed on 23/11/2007.
38. Chapin, S.J., et al., *Resource Management in Legion*, in *5th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP '99), in conjunction with the International Parallel and Distributed Processing Symposium (IPDPS '99).* 1999.
39. Natrajan, A., M.A. Humphrey, and A.S. Grimshaw, *Grid resource management in legion*, in *Grid resource management: state of the art and future trends.* 2004, Kluwer Academic Publishers. p. 145-160.
40. OSGA-WG, *The Open Grid Services Architecture, Version 1.0.* 2005, Global Grid Forum. p. 61.
41. Isaiadis, S. and V. Getov. *A Lightweight Platform for Integration of Mobile Devices into Pervasive Grids.* in *High Performance Computing and Communications, First International Conference, HPCC 2005.* 2005. Sorrento, Italy: Springer.
42. Millard, D.E., et al. *Experiences with Writing Grid Clients for Mobile devices.* in *Proceedings of 1st International ELeGI Conference on Advanced Technology for Enhanced Learning BCS Electronic Workshops in Computing (eWiC).* 2005. Vico Equense, (Napoli), Italy.

43. Phan, T., L. Huang, and C. Dulan. *Challenge: Integrating Mobile Wireless Devices Into The Computational Grid.* in *Proceedings of the 8th ACM International Conference on Mobile Computing and Networking (MobiCom).* 2002. Atlanta, Georgia, USA: ACM.
44. Kurkovsky, S., Bhagyavati, and A. Ray, *Modeling a Grid-Based Problem-Solving Environment for Mobile Devices.* Journal of Digital Information Management, 2004. **2, No 2**: p. 109-114.
45. Adzigogov, L., J. Soldatos, and L. Polymenakos, *EMPEROR: An OGSA Grid Meta-Scheduler Based on Dynamic Resource Predictions.* Journal of Grid Computing, 2005. **3**: p. 19-37.
46. Krauter, K., R. Buyya, and M. Maheswaran, *A taxonomy and survey of grid resource management systems for distributed computing*, in *Software-Practice and Experience (SP&E).* 2002. p. 135-164.
47. Berman, F.D., et al., *Application-level scheduling on distributed heterogeneous networks*, in *Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM).* 1996, IEEE Computer Society: Pittsburgh, Pennsylvania, United States.
48. Maheswaran, M., et al. *Dynamic Matching and Scheduling of a Class of Independent Tasks onto Heterogeneous Computing Systems.* in *8th Heterogeneous Computing Workshop (HCW '99).* 1999. San Juan, Puerto Rico: IEEE Computer Society Press.
49. Brauna, T.D., et al., *A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems.* Journal of Parallel and Distributed Computing, 2001. **61**(6): p. 810-837.
50. Silva, D.P.D., W. Cirne, and F.V. Brasileiro, *Trading Cycles for Information: Using Replication to Schedule Bag-of-Tasks Applications on Computational Grids*, in *Proc of Euro-Par 2003.* 2003: Klagenfurt, Austria.
51. Condor, *High Throughput Computing.* Last Accessed on 03/04/2008: http://www.cs.wisc.edu/condor/.
52. Kousalya, K. and P. Balasubramanie, *To Improve Ant Algorithm's Grid Scheduling Using Local Search.* International Journal of Computational Cognition, 2009. **7**(4): p. 47-57.

53. Choi, S., et al., *Volunteer Availability based Fault Tolerant Scheduling Mechanism in Desktop Grid Computing Environment*, in *Third IEEE International Symposium on Network Computing and Applications (NCA'04)*. 2004. p. 366-371.

54. Jin, H., et al., *Fault-tolerant grid architecture and practice.* Journal of Computer Science and Technology, 2003. **18**(4): p. 423-433.

55. Chandra, T.D. and S. Toueg, *Unreliable Failure Detectors for Reliable Distributed Systems.* Journal of the ACM (JACM), 1996. **43**(2): p. 225-267.

56. Favarim, F., et al., *Exploiting Tuple Spaces to Provide Fault-Tolerant Scheduling on Computational Grids*, in *10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'07)*. 2007, IEEE Computer Society. p. 403-411.

57. Hwang, S. and C. Kesselman, *A Flexible Framework for Fault Tolerance in the Grid.* Journal of Grid Computing, 2003. **1**: p. 251–272.

58. Lopes, R.F. and F.J.d.S.e. Silva, *Fault Tolerance in a Mobile Agent Based Computational Grid*, in *Sixth IEEE International Symposium on Cluster Computing and the Grid Workshops (CCGRIDW'06)*. 2006, IEEE. p. 8.

59. Favarim, F., et al., *GRIDTS: Tuple Spaces to Support Fault Tolerant Scheduling on Computational Grids*. 2006. p. 24.

60. Mujumdar, M., et al., *High Performance Computational Grids- Fault tolerance at System Level*, in *2008 First International Conference on Emerging Trends in Engineering and Technology*. 2008, IEEE Computer Society. p. 379-383.

61. Abramson, D., et al., *Simplified Grid Computing through Spreadsheets and NetSolve*, in *Proceedings of the High Performance Computing and Grid in Asia Pacific Region, Seventh International Conference on (HPCAsia'04) - Volume 00*. 2004, IEEE Computer Society.

62. Beguelin, A., E. Seligman, and P. Stephan, *Application Level Fault Tolerance in Heterogeneous Networks of Workstations.* Journal of Parallel and Distributed Computing, 1997. **43**: p. 147-155.

63. Geist, G.A., J.A. Kohl, and P.M. Papadopoulos, *PVM and MPI: a Comparison of Features.* Calculateurs Paralleles, 1996. **8**: p. 137-150.

64. Tannenbaum, T., et al., *Condor: a distributed job scheduler*, in *Beowulf cluster computing with Linux*. 2001. p. 307-350.

65. Andrade, N., et al., *OurGrid: An Approach to Easily Assemble Grids with Equitable Resource Sharing*, in *Job Scheduling Strategies for Parallel Processing*. 2003, Springer Berlin / Heidelberg. p. 61-86.

66. Casanova, H. and J. Dongarra, *NetSolve: A Network Server for Solving Computational Science Problems.* The International Journal of Supercomputer Applications and High Performance Computing, 1995: p. 212-223.

67. Sterck, H.D., et al., *A Lightweight Java Taskspaces Framework for Scientific Computing on Computational Grids*, in *2003 ACM symposium on Applied computing*. 2003, ACM: Melbourne, Florida. p. 1024-1030.

68. Apers, P.M.G. and G. Wiederhold, *Transaction Classification to Survive a Network Partition*. 1983, Stanford University: Stanford. p. 26.

69. Melliar-Smith, P.M. and L.E. Moser, *Surviving Network Partitioning.* Computer, 1998. **31**(3): p. 62-68.

70. Luckow, A. and B. Schnor, *Migol: A fault-tolerant service framework for MPI applications in the grid.* Future Generation Computer Systems, 2008. **24**(2): p. 142-152.

71. M.A.Rajan, et al., *A Study on Network partition Detection Relevant to Ad-hoc Networks: Connectivity Index Approach.* IJCSNS International Journal of Computer Science and Network Security, 2008. **8**(6): p. 150-158.

72. Iamnitchi, A. and I. Foster, *A Peer-to-Peer Approach to Resource Location in Grid Environments*, in *Grid Resource Management*, J. Weglarz, et al., Editors. 2003, Kluwer Publishing.

73. Gradwell, P., *Overview of Grid Scheduling Systems*. Last accessed on 01/04/2008: Department of Computer Science, University of Bath. http://peter.gradwell.com/phd/writings/computing-economy-review.pdf. p. 11.

74. Nimrod-G, *The Gridbus Project*. Last accessed on 15/04/2008: http://www.gridbus.org.

75. GrADS, *Grid Application Development Software* Last Accessed on 20/04/2008: http://www.hipersoft.rice.edu/grads/.

76. Madureira, A., J. Santos, and N. Gomes, *Hybrid Multi-agent System for Cooperative Dynamic Scheduling Through Meta-Heuristics*, in *Proceedings*

of the Seventh International Conference on Intelligent Systems Design and Applications. 2007, IEEE Computer Society.

77. Dimakopoulos, V.V. and E. Pitoura, *A Peer-to-Peer Approach to Resource Discovery in Multi-agent Systems*, in *Cooperative Information Agents VII*. 2003, Springer Berlin / Heidelberg. p. 62-77.
78. Czajkowski, K., et al., *Grid Information Services for Distributed Resource Sharing*, in *Tenth IEEE International Symposium on High-Performance Distributed Computing (HPDC-10)*. 2001, IEEE Press.
79. Leach, P., M. Mealling, and R. Salz, *A Universally Unique IDentifier (UUID) URN Namespace*. 2005, The Internet Engineering Task Force (IETF): http://www.ietf.org/rfc/rfc4122.txt.
80. Droms, R. *Dynamic Host Configuration Protocol*. RFC 2131 1997 [cited 2010 07/06/2010]; Available from: http://tools.ietf.org/html/rfc2131.
81. Lann, G.L. *Distributed Systems - Towards a Formal Approach*. in *Information Processing 77*. 1977.
82. Fokkink, W. and J. Pang. *Simplifying Itai-Rodeh Leader Election for Anonymous Rings*. in *4th Workshop on Automated Verification of Critical Systems - AVoCS'04*. 2005. London, UK.
83. Vasudevan, S., J. Kurose, and D. Towsley, *Design and Analysis of a Leader Election Algorithm for Mobile Ad Hoc Networks*, in *12th IEEE International Conference on Network Protocols (ICNP'04)*. 2008, IEEE: Berlin, Germany.
84. Dijkstra, E.W., *Self-stabilizing systems in spite of distributed control.* Communications of the ACM, 1974. **17**(11): p. 643-644.
85. Hatzis, K.P., et al., *Fundamental Control Algorithms in Mobile Networks*, in *SPAA '99*. 1999, ACM: Saint Malo, France.
86. Gallager, R.G., P.A. Humblet, and P.M. Spira, *A Distributed Algorithm for Minimum-Weight Spanning Trees.* ACM Transactions on Programming Languages and Systems, 1983. **5**(1): p. 66-77.
87. Kim, T.W., et al., *A leader election algorithm in a distributed computing system*, in *Fifth IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*. 1995: Cheju Island, South Korea. p. 481-485.

88. Shi, X.-b., J.-m. Gao, and L. Du, *A coordinator election algorithm for P2P MMOG*, in *5th International Conference on Visual Information Engineering, 2008. VIE 2008*. 2008, IEEE: Xian, China. p. 265-270.

89. Ambler, S.W. *Introduction to Concurrency Control*. Techniques for Successful Evolutionary/Agile Database Development 2006 [cited 2010 10/03/2010]; Available from: http://www.agiledata.org/essays/concurrencyControl.html.

90. Zhu, M., J. Bian, and W. Wu, *A novel collaborative scheme of simulation and model checking for system properties verification* Computers in Industry - Collaborative Environments for Concurrent Engineering Special Issue 2006. **57**(8-9): p. 752-757.

91. EDGSim. *EDGSim: Simulating the European Data Grid.* 2002 05/11/2002 [cited 14/01/2009]; Available from: http://www.hep.ucl.ac.uk/~pac/EDGSim/edgsim.html.

92. Schriber, T.J. and D.T. Brunner, *Inside Discrete-Event Simulation Software: How It Works and Why It Matters*, in *Winter Simulation Conference*, M.E. Kuhl, et al., Editors. 2005, IEEE. p. 11.

93. SIMGRID. *SimGrid.* [cited 14/01/2009]; Available from: http://simgrid.gforge.inria.fr/.

94. Buyya, R. and M. Murshed, *GridSim: a toolkit for the modeling and simulation of distributed resource management and scheduling for Grid computing.* Concurrency and Computation: Practice and Experience, 2002. **14**: p. 1175-1220.

95. Dumitrescu, C.L. and I. Foster, *GangSim: A Simulator for Grid Scheduling Studies*, in *CCGrid 2005*. 2005, IEEE: Cardiff, UK. p. 8.

96. Linington, P., *A Basic Simulation Package*, University of Kent. p. 5.

97. Linpack, *Linpack Benchmark -- Java Version*, in *http://www.netlib.org/benchmark/linpackjava/, last accessed on 14/06/2007.*

98. Mysaifu, *Mysaifu JVM*, in *http://www2s.biglobe.ne.jp/~dat/java/project/jvm/index_en.html, last accessed on 14/06/2007*. 2007.

99. SableVM, *The SableVM Project*, in *http://www.sablevm.org/, last accessed on 20/10/2007*. 2005.

100. Weiss, A.R., *Dhrystone Benchmark - History, Analysis, "Scores" and Recommendations, White Paper*. 2002, ECL, LLC. p. 16.

101. C-Creators. *Dhrystone benchmark written in Java*. [cited 14/12/2008]; Available from: http://www.c-creators.co.jp/okayan/DhrystoneApplet/.

102. IETF, *RFC 793: Transmission Control Protocol*, in *Protocol Specification*. 1981, IETF: http://www.ietf.org/rfc/rfc793.txt, last accessed on 11/12/2008.

103. Postel, J., *User Datagram Protocol*, in *RFC 768*. 1980, IETF: http://www.ietf.org/rfc/rfc768.txt, last accessed on 11/12/2008. p. 3.

104. Sulistio, A., et al. *A Toolkit for Modelling and Simulation of Data Grids with Integration of Data Storage, Replication and Analysis*. 2006 [cited 2010 01/02/2010]; Available from: http://www.gridbus.org/reports/datagrid_fgcs.pdf.

105. Paxson, V. and M. Allman, *Computing TCP's Retransmission Timer*, in *RFC 2988*. 2000, IETF: http://www.ietf.org/rfc/rfc2988.txt, last accessed on 11/12/2008. p. 8.

106. Karn, P. and C. Partridge, *Improving Round-Trip Time Estimates in Reliable Transport Protocols*. ACM SIGCOMM Computer Communication Review, 1995. **25**(1): p. 66 - 74.

107. Cowell-Shah, C.W. *Nine Language Performance Round-up: Benchmarking Math & File I/O*. 2004 [cited 13/01/2009]; Available from: http://www.osnews.com/story/5602/Nine_Language_Performance_Round-up_Benchmarking_Math_File_I_O/page2/.

108. MinGW. *Minimalist GNU for Windows*. [cited 2008 11/12/2008]; Available from: http://www.mingw.org/.

109. UPPAAL. *UPPAAL*. [cited 2010 15/03/2010]; Available from: http://www.uppaal.com/.

110. Berhmann, G., A. David, and K.G. Larsen, *A Tutorial on Uppaal.*, in *Formal Methods for the Design of Real-Time Systems*. 2004, Springer Berlin / Heidelberg. p. 200-236.

111. Behrmann, G., A. David, and K. Larsen. *A tutorial on Uppaal.* in *SFM-RT 2004, LNCS 3185*. 2004: Springer.

112. Bengtsson, J. and W. Yi, *Timed automata: Semantics, algorithms and tools*. Lecture Notes on Concurrency and Petri Nets, LNCS 3098, ed. W. Reisig and G. Rozenberg. 2004: Springer.

113. Milner, R., *A Calculus of Communicating Systems*. LNCS 92. 1980: Springer-Verlag.

114. Mascolo, C., L. Capra, and W. Emmerich, *Principles of mobile computing middleware*, in *Middleware for Communications*, Q. Mahmoud, Editor. 2004, John Wiley. p. 261-280.

115. Burke, R., *Hybrid Recommender Systems: Survey and Experiments.* User Modeling and User-Adapted Interaction, 2002. **12**: p. 331-370.

116. O'Connor, M., et al. *PolyLens: A Recommender System for Groups of Users*. in *Euro. Conf. on Computer Supported Cooperative Work (ECSCW)*. 2001. Bonn, Germany.

117. Breese, J.S., D. Heckerman, and C. Katie. *Empirical analysis of predictive algorithms for collaborative filtering*. in *Proceedings of the 14th Annual Conference on Uncertainty in Artificial Intelligence*. 1998. University of Wisconsin Business School, Madison, Wisconsin, USA: Morgan Kaufmann.

118. Blanco-Fernàndez, Y., et al. *AVATAR: a multi-agent TV recommender system using MHP applications*. in *The 2005 IEEE International Conference one-Technology, e-Commerce and e-Service, 2005. EEE '05*. 2005. Hong Kong.

119. Connor, M. and J. Herlocker, *Clustering items for collaborative filtering*, in *SIGIR-2001 Workshop on Recommender Systems*. 2001: New Orleans, Louisiana, USA.

120. Rashid, A.M., et al. *ClustKNN: A Highly Scalable Hybrid Model- & Memory-Based CF Algorithm*. in *WEBKDD 2006*. 2006. Philadelphia, Pennsylvania, USA.

121. Vozalis, E. and K.G. Margaritis. *Analysis of Recommender Systems' Algorithms*. in *In. The 6th Hellenic European Conference on Computer Mathematics & its Applications*. 2003. Athens, Greece.

122. Golub, G.H. and C.F.V. Loan, *Matrix Computations*. Vol. 1. 1983: The Johns Hopkins University Press. 476.

123. Grossberg, S., *Adaptive Resonance Theory*, in *Encyclopedia of Cognitive Science*, A67, Editor. 2000, Boston University: Boston, MA. p. 13.

124. Levenick, J., *A Welcome Change From Back-propagation Models of Cognition.* Psycoloquy, 1993. **4**(35).

125. Coulson, G., et al., *Towards a component-based middleware framework for configurable and reconfigurable grid computing*, in *Proceedings of the 13th*

IEEE International Workshops on Enabling Technologies: Infrastructure for collaborative Enterprises (WET ICE'04). 2004.

126. McKnight, L.W., J. Howison, and S. Bradner, *Wireless Grids - Distributed Resource Sharing by Mobile, Nomadic, and Fixed Devices.* IEEE Internet Computing, 2004. **8**(4): p. 2-10.
127. Hoheisel, A. and U. Der, *An XML-Based Framework for Loosely Coupled Applications on Grid Environments*, in *Computational Science – ICCS 2003 - Lecture Notes in Computer Science*. 2004, Springer Berlin / Heidelberg: Berlim, Germany. p. 664-665.
128. Kailash, S., et al. *Semantic Resource Description for Grid.* in *First Asia International Conference on Modelling & Simulation (AMS'07)*. 2007: IEEE Computer Society.
129. Kozierok, C.M. *The PC Guide*. Hard Disk Internal Performance Factors 2001 17/04/2001 [cited 10/01/2009]; Available from: http://www.pcguide.com/ref/hdd/perf/int_Rate.htm.

APPENDIX A – Recommender Systems

Recommender systems are designed to help users taking decisions in several different kinds of activity such as: choosing a restaurant, a place to visit, music to listen to inside a gym, a movie or any other TV content, for example. Many researchers have studied this kind of system, and some techniques have been developed with the objective of achieving better performance (typically in terms of accuracy and scalability).

In terms of the data used by a recommender system, we can say that, according to [115], they have:

- Background data: information known by the system before the recommendation process starts. E.g. the items to be recommended and the target users.
- Input data: information that must be sent to the system in order to compute the recommendation. E.g. the user's preferences and rankings for the items.
- An algorithm to combine the background and input data to make suggestions to the users.

Based on [2, 115, 116], we can classify the recommender techniques in the following six groups: (1) Collaborative Filtering, (2) Content-based, (3) Demographic, (4) Utility-based, (5) Knowledge-based and (6) Hybrid.

The *Collaborative Filtering* (CF) technique consists of finding users with similar preferences to the target user, and then computing recommendations based on these commonalities. This is the most often implemented and most mature of the techniques. Systems that use CF can be either memory-based, wherein the recommendation is made by comparing users directly to each other using correlations or other measures, or model-based, in which we first create a model (offline mode) and then run the model to get the recommendation (online mode) [116, 117]. Building a model may take hours or days, with the goal to make it small, fast and accurate [116]. For the model building process, several methods have been used, including Bayesian networks [117, 118], dimensionality reduction [5, 6] and clustering [119, 120], for example.

The *Content-based* method works by recommending items to the user, based on historical information, i.e. based on items that the user liked in the past [2, 115]. Systems that use this method learn a user's preferences based on the characteristics of the content that this user rated, and a profile is defined. The type of this profile varies according to the learning method adopted (e.g. decision trees, neural networks, vector-based).

Demographic techniques categorize the user based on personal attributes and make recommendations according to demographic classes [115]. This categorization is done by questioning the user to gather the personal data needed. In contrast to CF and content-based, demographic methods have the benefit of not requiring a history of user ratings.

The *Utility-based* method uses a utility function for each user to compute recommendations based on how useful the item is for the user. The main objective is to match user need to item utility.

Knowledge-based recommendation systems make inferences about user's needs and preferences and suggest an item. Knowledge-based approaches have knowledge about how a particular item meets a particular user need and can infer the relationship between a need and a possible recommendation [115].

The *Hybrid* approach consists of combining two or more methods mentioned above to gain better performance and try to avoid the problems or limitations presented by the individual use of the methods.

Assuming **I** as the set of items that can be recommended, **U** as the group of users whose preferences are known, **u** as the target user for the recommender system (i.e. the one to whom the system will suggest an item) and **i** as an item for which we would like to predict **u**'s preference, Table 10[7] summarizes the recommendation techniques presented previously.

All the recommender techniques presented earlier have their strengths and weaknesses, as we can see in Table 11[7]. The *new user ramp-up problem* is related to the loss of accuracy when recommending an item to a new user, since this user has not rated enough items to make a good suggestion. The *new item ramp-up* problem is similar to the new user problem, but related to items; since the new item has not been rated by enough users for a accurate recommendation, this item may never be

[7] Table extracted from 115. Burke, R., *Hybrid Recommender Systems: Survey and Experiments.* User Modeling and User-Adapted Interaction, 2002. **12**: p. 331-370.

suggested. *Gray sheep* [121], also known as the *Unusual User Problem*, refers to those users with peculiar preferences that are difficult to match consistently with preferences of any group of people.

Table 10 - Recommendation Techniques.

Technique	Background	Input	Process
Collaborative	Ratings from **U** of items in **I**.	Ratings from **u** of items in **I***.	Identify users in **U** similar to **u**, and extrapolate from their ratings of **i**.
Content-based	Features of items in **I**.	Ratings from **u** of items in **I***.	Generate a classifier that fits **u**'s rating behaviour and use it on **i**.
Demographic	Demographic information about **U** and their ratings of items in **I**.	Demographic information about **u**.	Identify users that are demographically similar to **u**, and extrapolate from their ratings of **i**.
Utility-based	Features of items in **I**.	A utility function over items in **I** that describes **u**'s preferences.	Apply the function defined by **u** to the items and determine **i**'s rank.
Knowledge-based	Features of items in **I**. Knowledge of how these items meet a user's needs.	A description of **u**'s needs or interests.	Infer a match between **i** and **u**'s need.

** The user **u** does not need to rate all the items in **I**, but the more items are rated the better is the accuracy of the recommendation. Normally some technique is used to reduce the sparsity of the matrix (i.e. to fill the non-rates spaces in the matrices), as presented later in this section.*

The drawback "L. Quality dependent on large historical data set" in Table 2 is related to the sparsity of the user-item matrices, i.e., lots of items with no rating. The sparser the matrices, the weaker are the recommendations. In [121] we can find some techniques that have been proposed to reduce sparsity like, for example: default Voting, pre-processing using average, use of *filterbots* and use of dimensionality reduction techniques.

One of the dimensionality reduction techniques is the Singular Value Decomposition (SVD) [5-8], a well-known matrix factorization method that factors and m x n matrix R into three matrices as following:

$$R = U \cdot S \cdot V'$$

Table 11 - Strengths and Weaknesses of recommendation techniques

Technique	Strengths	Weaknesses
Collaborative	A. Can identify cross-genre niches. B. Domain knowledge not needed. C. Adaptive: quality improves over time. D. Implicit feedback sufficient.	I. New user ramp-up problem. J. New item ramp-up problem. K. 'Gray sheep' problem. L. Quality dependent on large historical data set. M. Stability vs. Plasticity dilemma.
Content-based	B, C, D	I, L, M
Demographic	A, B, C	I, K, L, M N. Must gather demographic information.
Utility-based	E. No ramp-up required. F. Sensitive to change of preference. G. Can include non-product features.	O. User must input utility function. P. Suggestion ability static (does not learn).
Knowledge-based	E, F, G H. Can map from user needs to products.	P Q. Knowledge engineering required.

Where U and V are orthogonal matrices of size m x r and n x r respectively (r is the rank of R), and S is an r x r matrix with its diagonal entries having all singular values of R. These matrices are useful because of a property of SVD, which provides the best lower rank approximations of the original matrix R, in terms of Frobenius norm [6, 122]. Experiments with SVD have provided good performance in terms of accuracy, outperforming the traditional CF in some cases as presented in [6], when used with the Latent Semantic Indexing (LSI) technique.

The problem with SVD is that it is expensive computationally to factorize a m x n matrix, with a complexity in the order of $O((m+n)^3)$ [6, 34]. Nevertheless, SVD is efficient in terms of storage, in the order of $O(m+n)$.

The "Stability vs. Plasticity Dilemma" [123, 124] represented by the drawback "M" in Table 2 is a characteristic of learning systems, in which such systems may learn correct outputs to a set of inputs, but after learning to handle new subsets of inputs they may fail to respond appropriately to the previously (old) learned input.

Gathering demographic information ("N") is a disadvantage because users do not want to be disturbed by having to fill forms about their data, which in the TV world is more complicated, if they have to use a remote control to complete the task. The same problem occurs in asking the user to input a utility function ("O"), as in utility-based methods.

Utility and knowledge-based methods are not able to learn from previous recommendations ("P"), which make these methods incapable of 'discovering' a user niche like the collaborative systems do. Another problem regarding the knowledge-based methods is that they need knowledge acquisition ("Q. Knowledge engineering required"), which may not be available or difficult to get. According to [115], this information is classified as: (i) catalog knowledge, that refers to the objects/items being recommended; (ii) functional knowledge, that specifies how the user needs are mapped to the available items; and (iii) user knowledge, some information about the user for promoting better quality of recommendation.

A good option to try to overcome the disadvantages of these techniques is the use of hybrid methods. Hybrid methods usually provide better results in terms of accuracy. A classification for hybrid methods and some examples can be found in [115], and some of the combinations still need further investigation.

APPENDIX B – Resource Description

This appendix presents our proposed resource description standard to be used by the *Description Handler* component described in the section 3.5.

Independently of the kind of grid (wired or wireless), standardized mechanisms are required for describing resources available on the connected devices and without them, resource sharing would become impractical if not impossible [10, 125].

According to [126], none of the existing schemes provided for resource description covers all resources, but when taken together they can define most of the shared resources. It is impossible to define an initial schema that will cover all the resources that can be found in the home environment, since new technologies/devices (and consequently, new resources) are still to come. However, our first requirement that can be set for the resource description standard is that it must be extendable to reflect the new resources available in the grid.

Similarly to the Legion system [37], we are going to adopt a object-oriented model for resource description, which suits the requirement for extensibility of the data model. With this abstract model, we can describe the resources without referring to a specific system or language. In Figure 93, we present a diagram with objects that represent the kind of resources found in the home environment and their attributes.

In our model, we propose the object ResourceDescriptor that is the generalization for all kinds of resource. It is composed of three elements: the identification (ID) of the resource, a small description and an access policy.

The approach adopted for the issue of IDs in this thesis is presented in Chapter 3.

Since each device has control over its own resources, we need a way to identify how these resources can be used and the users/devices allowed access to them. We can do it by using an access policy, which is an important component of resource description. We are going to talk more about this policy later, when specifying the security requirements.

Each resource can be associated with any other resource. To represent this we have defined the association class ResourceAssociation, which contains the type of relationship between resources. The types supported in our model are: depends, provides, conflicts and suggests. This classification was based on the work presented in [127], and will be useful for the resource discovery system.

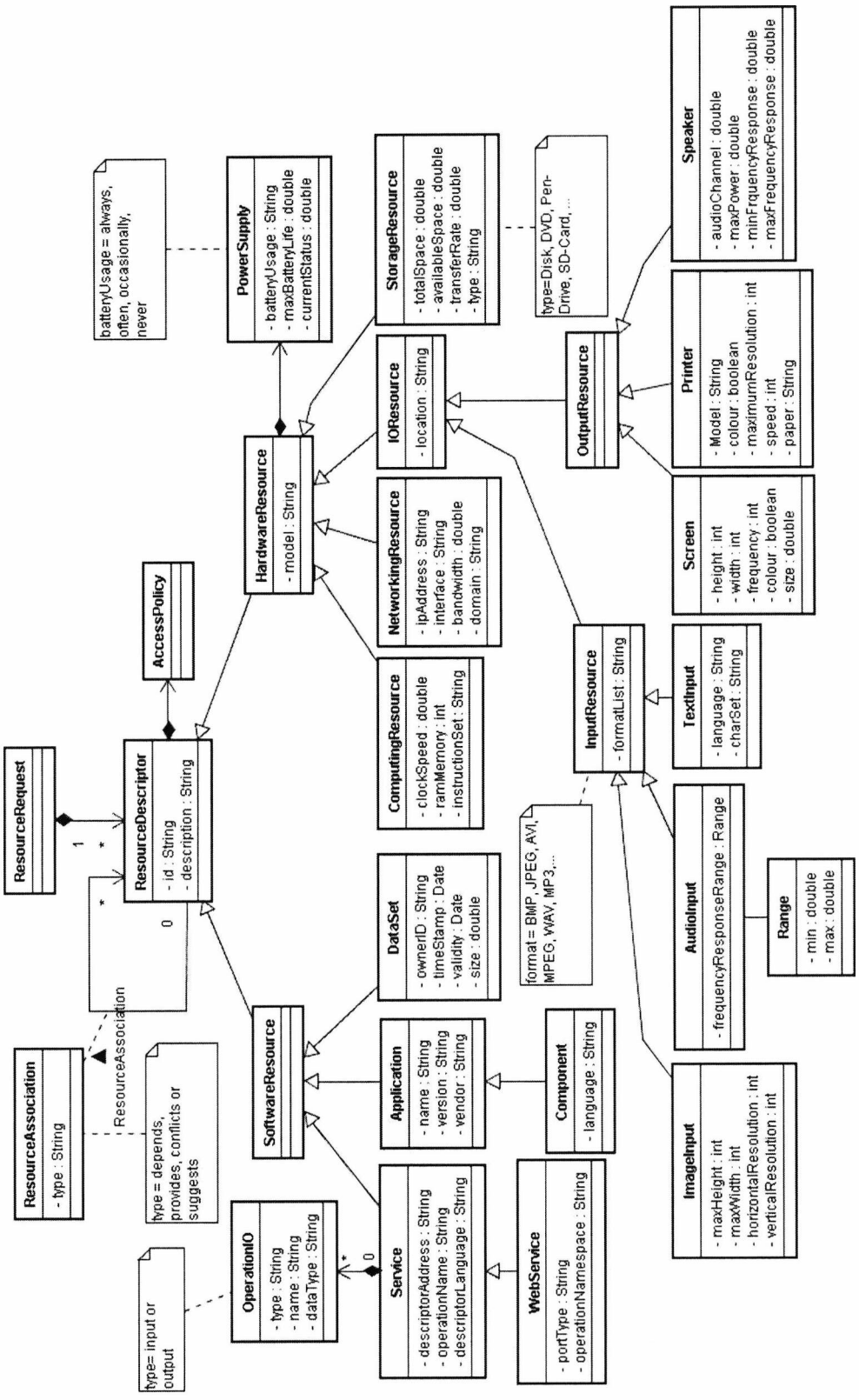

Figure 93 - Resource description diagram

- Depends: indicates that one resource depends on other resources to run properly. As an example we can mention a component written in Java that requires a Java Virtual Machine (JVM);
- Provides: specifies which other resources a certain resource may have available for sharing. For example, a computer that can provide disk storage or MS Windows as an operating system;
- Conflicts: refers to resources that are not able to be used together. For example, the operating system Symbian for mobile devices conflicts with a hardware resource that implements the x86-64 instruction set architecture;
- Suggests: recommends that a certain resource be used with another. An example would be Java applications or components for PDAs, where the description could recommend the use of a specific JVM implementation that could give better results.

SoftwareResource and HardwareResource

Extending the ResourceDescriptor object, we defined two main classes of resources: SoftwareResources and HardwareResources. This classification was based on [127], which describes an XML Schema for resource description (Grid Resource Definition Language - GResourceDL), supporting six basic resource types: *concrete software components*, *software classes*, *concrete hardware resources*, *hardware classes*, *concrete data*, and *data classes*. The *concrete software component* and *software class* types in GResourceDL are represented by the objects SoftwareResource, Application and Component in our model, where the first object represents any tool or operating system, being characterized by its name, version and vendor; and the last extends the Application by indicating the programming language used to develop it.

The *concrete hardware resource* and *hardware class* types in GResourceDL are represented by the HardwareResource object in our proposal, which was extended based on the grid resource class hierarchy presented in [128], which proposes an ontology for the grid resource description. That work classifies grid resources as *computing resources*, *network resources* and *storage resources*, which are represented in our model by the objects ComputingResource, NetworkResource and StorageResource, respectively.

Computing Resource

ComputingResource objects describe the resource's processing capabilities, like the CPU's clock speed (e.g. 2.0 GHz), the amount of RAM memory available (e.g. 1024 MB), and the instruction set architecture supported (e.g. ARM, MIPS, PowerPC, x86, x86-64).

NetworkResource

The NetworkResource object provides information about network capabilities: IP address (e.g. 192.168.2.5), communication interface (e.g. Ethernet, 802.11g, Bluetooth, ATM), bandwidth (e.g. 100.00 Mbps, 0.056 Mbps), and domain (e.g. "SensorNetwork", "HomeNetwork").

StorageResource

The StorageResource object is related to the storage capacity available for sharing in the grid: the type of the media (e.g. Disk, DVD, Pen-drive, SD-Card), total and available space (e.g. 120,000.00 MB), and internal data transfer rate (the rate at which data is read from the physical device and transferred to the internal cache or read buffer [129], e.g. 10 MB/second).

IOResource

For the remaining hardware devices we can find in the home environment like cameras, scanners, printers, keyboards and TV sets, for example, we have defined the object IOResource. This object includes an identification of the location in the house, which can be useful to determine the closest device to the user when his/her location is also available. We have defined two sub-objects for IOResource: InputResource and OutputResource.

InputResource

This object represents devices that users can utilize to input data to some grid application, when required. The InputResource object has a list of formats (formatList) as attribute, which corresponds to the standard used to store the data like, for example, a webcam that can generates images in "AVI" or "MPEG". This list is a string composed by the name of the supported formats (AVI, WAV, MP3, etc.), all separated by ",".

We have also defined extensions to the InputResource with the objective of getting more specific information about certain types of input devices: AudioInput, ImageInput and TextInput. The first object describes information about the audio capabilities of a device that can capture audio, like a microphone, for example; its attribute is the frequency response range (e.g. 20 Hz to 2000 Hz).

The ImageInput object represents information about the maximum quality of image that can be acquired by devices that can capture this kind of data, like a camera or scanner, for example. The attributes that compose this object are the maximum height and width (e.g. 8.5 inches), and the maximum horizontal and vertical resolution (e.g. 1200 dpi) that a generated image can have. In this thesis we use image as a general term for static (e.g. figures) and dynamic (e.g. videos) images.

Devices that can be used to input information in text format (e.g. a keyboard) are represented by the TextInput object, which is composed by the language (e.g. "English") and the char set (e.g. "ASCII") supported by the resource.

OutputResource

This object represents devices with capability to output data. We have categorized the different devices according to the type of output it produces.

The first category is represented by the Screen object, which corresponds to devices that show information in a monitor/screen like, for example, a TV set or a computer monitor. The information that describes these devices are: the height and width that a certain device can display (e.g. 800 pixels), the frequency in which the images are updated (e.g. 60 Hz) and the information to indicate if the device supports colours or not.

The second category is described by the Printer class, which contains information about printers like the model (e.g. "HP Deskjet 680C"), a boolean value indicating if the printer supports coloured printing, the maximum resolution (e.g. 600 dpi), the speed (e.g. 6 pages per second) and the kind of paper it has (e.g. "A4").

The last category consists of devices that can emit sound, and are described by the Speaker object. The attributes of this object are: the audio channel type (e.g. "5.1"), the max power (e.g. 30 W), the minimum and the maximum frequency response (e.g. 18000 Hz).

DataSet

The last two resource types in GResourceDL (*concrete data* and *data classes*) were mapped into the DataSet object in our model. This object plays an important role in the development of a data grid. It describes the data owner identification (depending on the authentication model adopted, e.g., it could be a X.500 Subject Name), the time stamp (the date and time the data was created or stored in a certain place), an expiry date for the data (an empty/null date means that the data will not expire), and the size of the date (e.g. 50 MB). The size is important information in case the device where the data is stored needs to make a planned shut down and need to find another device with the capabilities and permission to store it permanently or temporarily, until the date expressed by the validity field. The data location is expressed using the URI inherited from the SoftwareResource object.

Service

Some devices may also share resources as services, and for that reason we have defined the object Service that extends SoftwareResource. The Service object is composed of the URI for the service (inherited from SoftwareResource object), the operation name, the address where the service description can be found, and the language in which it is expressed (e.g. "WSDL" for web services description, and "IDL" for denoting CORBA services). The Service also contains the description of the operation's parameters and result, described in our model through the OperationIO object, which represents the type of the element (if "input", i.e. a parameter, or "output", i.e. the return), the name of it, and the data type (e.g. integer, String, boolean). The Service object can be extended to allow the definition of specific attributes for the service invocation. In our model we have defined the WebService extension for web services invocation, which denotes the port type and the namespace for the operation.

PowerSupply

Since some of the devices in the home grid may use batteries as power supplier, we need to define which of them have this characteristic and be able to envisage how much power is still remaining, so we can provide a better fault tolerance mechanism. For this reason we define the PowerSupply object, which indicates the battery usage from the device (if it "always" uses a battery, or if it uses one "often", or

"occasionally", or "never"), the maximum battery life (e.g. 5 hours), and the current status of the battery (e.g. 50.7% of usage). A PowerSupply object is defined as an attribute of hardware resources.

ResourceRequest

When a grid application needs to find resources for running its jobs, it will have to create an instance of ResourceRequest object, and define a list of ResourceDescriptors that represents the minimum requirements for running such jobs. We are going to talk about ResourceRequest in more detail when specifying the resource discovery system.

APPENDIX C – Pseudo-code for the proposed solution

In this appendix, we aggregate the pseudo-codes used in the Chapter 4 by their corresponding software component.

Consumer

```
sendJob(job) {
    send(JOB_SUBMISSION_REQUEST, scheduler, job);
    lastJobSent = job;
    startSubmissionTimer();
}

handleJobSubmissionResponse(jobLocalID, jobGlobalID) {
    if (jobLocalID IS lastJobSent.LOCAL_ID) {
        if (lastJobSent.status IS NOT COMPLETE) {
            lastJobSent.GLOBAL_ID = jobGlobalID;
            jobsSent.add(lastJobSent);
            broadcast(JOB_DESCRIPTION, lastJobSent.description);
            //start the timer for the job timeout
            startCompletionTimer(lastJobSent);
        } else {
            send(CANCEL_JOB, jobGlobalID);
        }
    } else {
        send(CANCEL_JOB, jobGlobalID);
    }
}

handleJobCompletion(job) {
    jobsSent.remove(job);
    jobsCompleted.add(job);
}

handleJobSubmissionTimeout() {
    //read application policy to
    //decide what to do: process the job locally
```

```
    //or resubmit the job
    if (POLICY IS RESUBMIT) {
        send(lastJobSent);
    } else if (POLICY IS EXECUTE) {
        execute(lastJobSent);
        lastJobSent.status = COMPLETE;
        jobsCompleted.add(job);
    }
}

startRecovery() {
    lastKnownID = jobsSent.lastJob.GLOBAL_ID;
    send(RECOVER_ID, scheduler, lastKnownID);
}

handleConnectionTimeout(message) {
    if (message IS JOB_SUBMISSION_REQUEST
            OR message IS CANCEL_JOB
            OR message IS RECOVER_ID) {
        coordinator.startElection();
    }
}
```

Scheduler

```
handleJobSubmissionRequest(job) {
    job.GLOBAL_ID = generateID(ID, lastIssuedID++);
    changeStatus(message.job, AVAILABLE);
    pendingJobs.add(job);
    send(JOB_SUBMISSION_RESPONSE, job.owner, job.GLOBAL_ID,
        job.LOCAL_ID);
}

handleScheduleRequest(queue, off_interval, message) {
    chosenJob = schedule(queue, queue.size, message.sender);
    jobsList = createEmptyList();
    //populates the jobsList with missing jobs and returns
    //TRUE if there are more missing jobs, or FALSE otherwise.
    more_flag = getMissingJobs(jobsList);
    send(SCHEDULE_RESPONSE, chosenJob, queue, jobsList, more_flag);
```

```
}

schedule(providerJobsList, n, providerID) {
    result = NULL;
    i = 0;
    while (i < n) {
        temp = pendingJobs.get(providerJobsList.next);
        if (temp IS NOT NULL) {
            if (temp.status IS NOT SCHEDULED) {
                if (result IS null) {
                    result = temp;
                    result.providerID = providerID;
                    result.status = SCHEDULED;
                }
            }
        } else {
            providerJobsList.remove(temp);
        }
        i = i + 1;
    }
    return result;
}

handleCancelJob(jobID) {
    pendingJobs.remove(jobID);
}

handleConsumerUnavailable(consumer) {
    removeAllJobsFromConsumer(consumer);
}

handleJobCompleted(jobID) {
    job = pendingJobs.remove(jobID);
    completedJobs.add(job);
}

handleRecoverID(jobID) {
    if (jobID > lastIssuedID) {
        lastIssuedID = jobID;
    }
}
```

```
handleRecoverQueue(pendingJobs, completedJobs) {
    handleRecoveredID(pendingJobs.lastJob);
    handleRecoveredID(completedJobs.lastJob);
    updatePendingJobs(pendingJobs);
    updateCompletedJobs(completedJobs);
}

handleConnectionTimeout(message) {
    if (message IS JOB_SUBMISSION_RESPONSE) {
        pendingJobs.remove(message.job);
    } else if (message IS SCHEDULE_RESPONSE) {
        changeStatus(message.job, AVAILABLE);
    }
}

handleRecoveryTimeout() {
    if (hasMoreRecoveryMessages() IS TRUE) {
        timer.startRecoveryTimeout(T);
    } else {
        broadcast(RECOVERY_COMPLETE);
    }
}

handleMerge(other_scheduler) {
    mergeUtility = computeMergeUtility();
    size = pendingJobs.size + completedJobs.size;
    mergeRandom = RANDOM(0, 1);
    send(MERGE_UTILITY, other_scheduler, mergeUtility,
        size, mergeRandom);
}

handleMergeUtility(message, received_utility, received_size,
        received_random) {
    size = pendingJobs.size + completedJobs.size;
    if (size > received_size) {
        mergeUtility = mergeUtility + 1;
    } else if (received_size < size) {
        received_utility = received_utility + 1;
    }
    if (mergeUtility IS received_utility) {
```

```
            if (mergeRandom IS received_random) {
                send(MERGE, message.sender, THIS);
            } else if (mergeRandom < received_random) {
                send(SYNC, pendingJobs, completedJobs);
                stopLocalScheduler();
            } else {
                startSyncTimer();
            }
        } else if (mergeUtility < received_utility) {
            send(SYNC, pendingJobs, completedJobs);
            stopLocalScheduler();
        } else {
            startSyncTimer();
        }
    }

    handleSyncMessage(pendingJobs, completedJobs) {
        stopSyncTimer();
        updatePendingJobs(pendingJobs);
        updateCompletedJobs(completedJobs);
        broadcast(MERGE_COMPLETE);
    }

    handleSyncTimeout() {
        broadcast(MERGE_FAILED);
    }

    handleConnectionTimeout(message) {
        if (message IS SYNC) {
            broadcast(MERGE_FAILED);
        }
    }

    handleMergeRecoverQueue(pendingJobs, completedJobs) {
        updatePendingJobs(pendingJobs);
        updateCompletedJobs(completedJobs);
    }
```

Provider

```
handleJobDescription(job) {
    if (job.scheduler IS currentScheduler) {
        local_status = coordinator.getStatus();
        processable = matchmake(local_status, job.DESCRIPTION);
        if (isProcessingJob AND job IS chosenJob) {
            if (processable IS FALSE) {
                stopExecutingJob();
                send(CHANGE_STATUS, scheduler, job.GLOBAL_ID);
            }
        } else {
            if (processable IS TRUE) {
                classifyAsProcesable(job);
            } else {
                classifyAsNotProcesable(job);
            }
            jobs.add(job);
            requestSchedule();
        }
    } else {
        send(MERGE, job.scheduler, currentScheduler);
        send(MERGE, currentScheduler, job.scheduler);
    }
}

requestSchedule() {
    if(hasProcessableJobs() && jobsSent IS FALSE) {
        queue = getFirstMProcessableJobs();
        off_interval = getOldestOffInterval();
        send(SCHEDULE_REQUEST, scheduler, queue, off_interval);
        jobsSent = TRUE;
    }
}

handleScheduleResponse(chosen_job, updated_queue
        missing_jobs, more_flag) {
    for (i = 0 .. missing_job.size) {
        job = missing_jobs[i];
        processable = matchmake(local_status, job.DESCRIPTION);
        if (processable IS TRUE) {
            classifyAsProcesable(job);
        } else {
```

```
            classifyAsNotProcesable(job);
        }
    }
    updateLocalQueue(updated_queue, missing_jobs);
      jobsSent = FALSE;
    if (more_flag IS FALSE) {
        removeOldestOffInterval();
    }
    if (chosen_job IS NOT NULL) {
        result = execute(chosen_job);
        //writes the result to the specified location
        writeResult(result, chosen_job.outputLocation);
        //informs the consumer of the job completion
        send(JOB_COMPLETION_CONSUMER, chosen_job.owner);
        //informs the scheduler of the job completion
        send(JOB_COMPLETION_SCHEDULER, scheduler,
            chosen_job.GLOBAL_ID);
        completedJobs.add(chosen_job);
    }
    requestSchedule();
}

startRecovery() {
    send(RECOVER_QUEUE, scheduler, jobs, completedJobs);
}

handleConnectionTimeout(message, job) {
    if (message IS JOB_COMPLETION_CONSUMER) {
        send(CONSUMER_UNAVAILABLE, scheduler, job.owner)
    } else if (message IS JOB_COMPLETION_SCHEDULER
            OR message IS SCHEDULE_REQUEST
            OR message IS CHANGE_STATUS
            OR message IS RECOVER_QUEUE
            OR message IS CONSUMER_UNAVAILABLE
            OR message IS MERGE_RECOVER_QUEUE) {
        coordinator.startElection();
    }
}

sendMergeRecoverQueue() {
    send(MERGE_RECOVER_QUEUE, currentScheduler,
```

```
        jobs, completedJobs);
}
```

Election Component

```
startElection() {
    startElection(SCHEDULER_FAILURE);
}

handleSchedulerFailureMessage(received_utility) {
    if (THIS IS currentScheduler) {
        broadcast(SCHEDULER_ELECTED, utility, THIS);
    } else {
        if (electionStarted IS FALSE) {
            if (schedulerCapable IS TRUE) {
                local_utility = computeUtility();
                if (local_utility > received_utility) {
                    broadcast(HIGHER_UTILITY, local_utility);
                    timer.start(T);
                    isCandidate = TRUE;
                } else {
                    isCandidate = FALSE;
                    if (timer.started IS TRUE) {
                        timer.stop();
                        timer.start(2*T);
                    }
                }
            } else {
                if (timer.started IS FALSE) {
                    timer.start(2*T);
                }
            }
            electionStarted = TRUE;
        }
    }
}

handleHigherUtility(received_utility, failedScheduler) {
    if (electionStarted IS TRUE) {
        if (schedulerCapable IS TRUE) {
```

```
            local_utility = computeUtility();
            if (local_utility > received_utility) {
                broadcast(HIGHER_UTILITY, local_utility);
                timer.start(T);
                isCandidate = TRUE;
            } else {
                isCandidate = FALSE;
                if (timer.started IS TRUE) {
                    timer.stop();
                    timer.start(2*T);
                }
            }
        } else {
            if (timer.started IS FALSE) {
                timer.start(2*T);
            }
        }
    }
}

handleElectionTimeout() {
    if (isCandidate IS TRUE) {
        broadcast(SCHEDULER_ELECTED, utility, THIS);
        coordinator.startRecoveryTimer();
    } else {
        startElection(RESTART_ELECTION);
    }
}

startElection(message) {
    utility = computeUtility();
    electionStarted = TRUE;
    broadcast(message, currentScheduler, utility);
    if (schedulerCapable IS TRUE) {
        timer.start(T);
        isCandidate = TRUE;
    } else {
        timer.start(2*T);
    }
}
```

```
handleSchedulerElected(senderUtility, newScheduler) {
    timer.stop();
    electionStarted = FALSE;
    if (currentScheduler IS NOT newScheduler) {
        if (THIS is currentScheduler) {
            if (utility IS senderUtility) {
                //generates a random number between
                //0 and 1 and adds it to the utility
                random = RANDOM(0, 1);
                utility = utility + random;
                broadcast(SCHEDULER_ELECTED, utility, THIS);
            } else if (utility > senderUtility) {
                broadcast(SCHEDULER_ELECTED, utility, THIS);
            } else {
                currentScheduler = newScheduler;
                coordinator.startRecovery();
            }
        } else {
            currentScheduler = newScheduler;
            coordinator.startRecovery();
        }
    }
}
```

Coordinator

```
handleMergeComplete(message) {
    currentScheduler = message.sender;
}

handleMergeFailed(message) {
    currentScheduler = message.sender;
    provider.sendMergeRecoverQueue();
}
```

APPENDIX D – C++ Code of DMS and OLB scheduling algorithms

The C++ code presented in this appendix was developed for counting the number of instructions required by the schedulers to be used in the simulations. The explanation of how we have computed the number of instruction from the C++ code is presented in the section 5.4.6 of this thesis.

Distributed Matchmake Scheduling (DMS)

```
#include <string>
#include <map>
#include <list>

using namespace std;

class Job{
  public:
      int id;
      bool isScheduled;
      int serverID;

      double cpu;
      double disk;
      double bandwidth;
      string os;
      bool battery;
};

class Provider{
  public:
      int id;

      double cpu;
      double disk;
      double bandwidth;
      string os;
      bool battery;
};
```

```
class Scheduler{
  public:
      map<int,Job> localJobs;
      void schedule(list<Job> &providerJobsList, int n, int
serverID, Job &result);
};

void Scheduler::schedule(list<Job> &providerJobsList, int n, int
serverID, Job &result) {
    int i = 0;
    list<Job>::iterator it = (providerJobsList).begin();
    bool resultIsSet = false;

    while (i < n) {
        map<int, Job>::iterator it1 = localJobs.find(i);

        if( it1 != localJobs.end() ) {

            Job temp = (*it1).second;
            if (! (temp.isScheduled)) {
               if (! resultIsSet) {
                  result = temp;
                  result.serverID = serverID;
                  result.isScheduled = true;
                  resultIsSet = true;
               }
            }

        } else {
            providerJobsList.erase(it);
            n = n - 1;
        }

        it++;
        i = i + 1;
    }
}
```

Optimistic Load Balance (OLB)

```
#include <stdlib.h>
#include <time.h>
#include <string>
#include <list>

using namespace std;

class Job{
  public:
      int id;
      bool isScheduled;
      int serverID;

      double cpu;
      double disk;
      double bandwidth;
      string os;
      bool battery;
};

class Provider{
  public:
      int id;

      double cpu;
      double disk;
      double bandwidth;
      string os;
      bool battery;
};

class Scheduler{
  public:
      list<Job> localJobs;
        list<Provider> providers;
      int m;   // number of servers available
      int M;  //total number of servers
      list<bool> readyServers;
      bool match(Job, Provider);
      void schedule(Job &job);
};
```

```
void Scheduler::schedule(Job &job) {
    Provider temp[m];
    int i = 0;
    int n = 0;

    list<Provider>::iterator it = (providers).begin();
    list<bool>::iterator it1 = (readyServers).begin();

    while (i < M) {
        if ((*it1) == true) {
            if (match(job, (*it))) {
               temp[i] = *it;
               n++;
            }
        }
        i = i + 1;
        it++;
        it1++;
    }

    if (n > 0) { // 2
        srand ( time(NULL) );
        int index = rand() % n;
        job.isScheduled = true;
        job.serverID = temp[index].id;
    }
}

bool Scheduler::match(Job job, Provider provider) {
     return ((job.cpu <= provider.cpu) & (job.disk <= provider.disk)
            & (job.bandwidth <= provider.bandwidth)
            & (job.os.compare(provider.os) == 0)
            & (job.battery == provider.battery));

}
```

APPENDIX E – Abstract Templates

Here we present the templates for the abstract model. The abstract provider template is presented in Figure 92, Chapter 7.

These abstract templates are results of our effort to generate a model that requires less computational resource to be verified. A proof that the abstract model can be used to represent the behaviour of the original concrete model is presented in the Chapter 7.

Abstract Consumer Template

The abstract model in Figure 94 represents an abstraction for the concrete model of the consumers' behaviour presented in Figure 87 (Chapter 7).

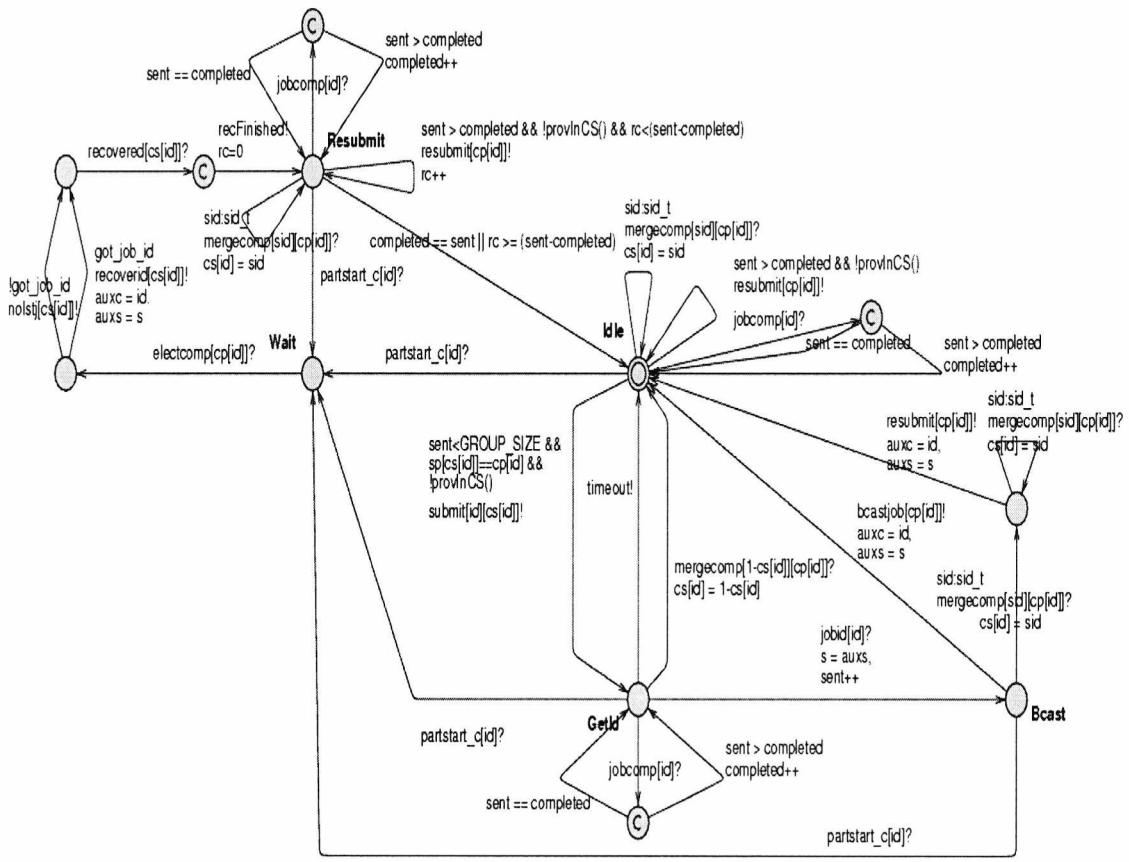

Figure 94 - Abstract Consumer Template.

Abstract Pmerger Template

The abstract model in Figure 95 represents an abstraction for the concrete model of the *Pmerger* component presented in Figure 89 (Chapter 7).

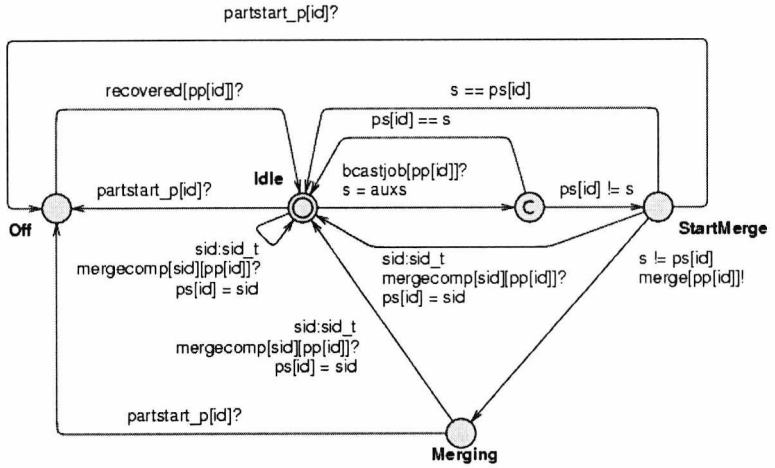

Figure 95 - Abstract Pmerger Template.

Abstract Scheduler Template

The abstract model in Figure 96 represents an abstraction for the concrete model of the schedulers' behaviour presented in Figure 90 (Chapter 7).

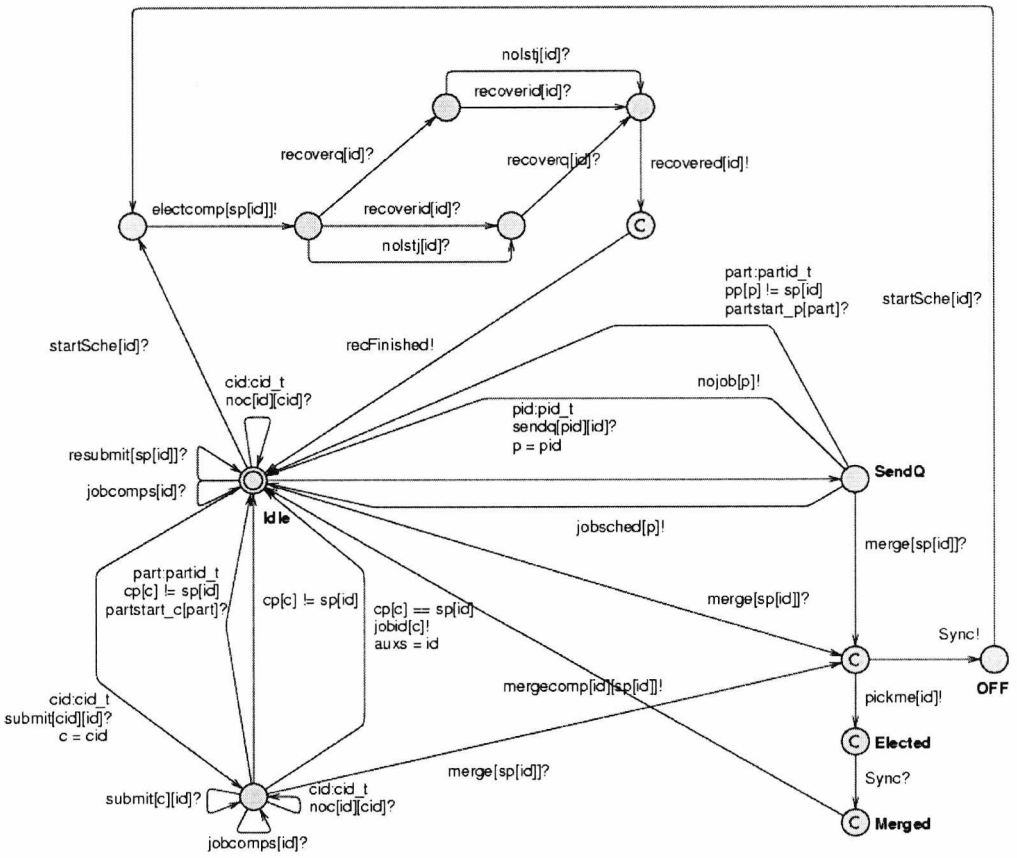

Figure 96 - Abstract Scheduler Template.