New Environments for parallel execution and simulations

João Nuno de Oliveira e Silva
(Mestre)

Proposta de tese de Doutoramento para apreciação pela Comissão de Acompanhamento de Tese

Orientador:	Doutor Luís Manuel Antunes Veiga
Co-Orientador:	Doutor Paulo Jorge Pires Ferreira

Dezembro de 2010

Resumo

O aumento de poder computacional dos computadores de secretária tem levado a crescimento de poder computacional passível de ser utilizado na execução de aplicações de simulação. Tal uso tem sido aplicado em projectos de computação distribuída: investigadores desenvolvem programas de simulação (por exemplo, na área da biologia computacional) que são executados nos computadores de secretária espalhados pela Internet.

Embora investigadores com projectos de execução longa e com grande visibilidade tenham a capacidade de captar os recursos disponíveis, outros utilizadores (domésticos ou com necessidades de cálculo pontuais) não conseguem de facto tirar proveito desses recursos. Estas novas categorias de utilizadores também têm necessidades de poder computacional.

Investigadores das áreas da estatística ou economia necessitam de efectuar *batches* de simulações, mas sem terem os recursos (horas de processamento e número de processadores) que outros investigadores têm (por exemplo da área da biologia computacional). O *rendering* de imagens ou animações também pode tirar proveito de computadores remotos. Utilizadores domésticos, ou designers gráficos, pontualmente têm necessidade de gerar imagens complexas, tarefas que poderão facilmente ser paralelizáveis e executadas em computadores remotos.

Estas novas classes de utilizadores usam aplicações genéricas (software de análise estatística, ou *renders* gráficos) para resolver os seus problemas, que podem facilmente ser paralelizados gerando tarefas independentes. No entanto, é difícil aceder a recursos computacionais para resolver eficientemente os seus problemas: i) é necessário a existência de alguma ligação institucional aos donos dos sistemas para computação de alto desempenho, ii) o acesso a recursos espalhados pela Internet, através de sistemas de computação distribuída, está limitada pela duração das simulações e visibilidade do trabalhos a resolver e iii) a criação de clusters para a execução *on-site* desses trabalhos é complexa.

Abstract

The computational power increase of desktop computers has made it a good source for execution cycles to be used in the simulation of physical phenomenon. This available computing power has been used in Distributed Computing projects: researchers develop simulation code (for instance in the area of computational biology) that is executed in desktop computers scattered over the Internet.

Although researchers with long running or high visibility projects are able to attract enough resources, other users (domestic or with limited computing needs) can not. These new classes of users are also in need of processing cycles to solve their problems.

For instance, researchers in the area of Statistics or economy need to make batches of simulation, but not being able to access to resources (processing time and number of processors) others have. In industrial environments the rendering of animations or images can also take advantage of remote computers. Domestic users, or graphical designers, at times, also need to render complex images, tasks that can easily take advantage of otherwise idle remote computers.

These new user classes use generic off the shelf applications (statistical software packages, or graphical renders) to solve their problems, that can be easily parallelized in independent tasks. On the other hand, it still is difficult to them to access the needed computational resources to efficiently solve their problems: i) there is a need to have an institutional relation with the owners of the high performance computing infrastructures, ii) the access to Internet scattered resources is limited by the duration and visibility of the jobs, and iii) the creation and management of computational clusters to on-site execution of the jobs is complex.

Palavras Chave

Programação paralela
Computação Distribuída
Computação na Nuvem partilha de ciclos

Keywords

Parallel Programming
Distributed Computing
Cloud Utility Computing Cycle sharing

Table of Contents

1 Introduction

In recent year there has been a trend on the upper scale *supercomputers*: instead of dedicated, specialized hardware, these new supercomputers use (close to) off the shelf hardware in their construction [41]. The more powerful supercomputers are built as a collection of independent computers, linked by a high speed network connections. Although these computers are much more powerful than desktop personal computers (in terms of processing power, storage capacity and reliability), their architecture and running software are essentially the same.

This similarity between commodity hardware and specialized high performance computing nodes has lead to the idea of using available personal computers as processing nodes, of a larger, more diffuse, distributed supercomputer.

On the user level, we can also see a higher demand for computational power. In the areas traditionally demanding computational power (e.g. physics and mechanics) scientists still require numerical simulation, but new areas are emerging also requiring the simulation of processes (e.g. economy, computational biology). Besides these new knowledgeable users, also on the commoner side there are new requirements for high levels of processing power to solve some of their problems, such as designers that need to do complex image renderings, hobbyists that need to render animations, or even users with batches of photos to process for enhancement.

These new users have different requirements from the traditional high performance computing users. While traditional ones require a dedicated computational infrastructures (due to the necessary synchronization between tasks) and have easy access to such infrastructures, other users do not. Most projects deployed by the new users classes fit into the Bag-of-Tasks problems (mainly composed of independent tasks, thus embarrassingly parallel), and these users can neither afford nor access dedicated computing infrastructures.

Researchers from the statistical or economy areas may require to perform batches of simulations without strict requirements, as the simulations are independent and decoupled between themselves. The rendering of frames of movies can also be easily paralyzed as independent tasks. Domestic or graphical industry users' needs also fit into the Bag-of-Tasks problems. The rendering of a image or processing of batches of images are composed of independent tasks.

The tools used by this new user class are most off-the-self applications (such as statistical analysis software or image renders). Although these tools are not designed to take advantage of the the

available parallel execution environments, a task decomposition of the problems can take advantage of these unmodified applications.

The similarity of software and hardware (in specialized infrastructures and desktop home computers) added to the high performance of personal computers may lead to the idea that common users (hobbyists or those with few computational requirements) may have easy access to the necessary computational power.

Although there are new sources of computational power, their availability to common users is still low, due to the inexistent tools and mechanisms adapted to these new user class.

The most accessible source of computing power are the desktop computers owned by each user. In some cases, when connected by a local area network, it is possible to use them as a cluster capable of speeding up existing computations.

The ubiquitous connection of these computers to the Internet also makes them a good source of computing cycles to remote users, as demonstrated on the various successful Distributed Computing projects.

Recently, a new source for computing power has appeared. Different form other cloud services, Amazon offers the possibility to dynamically switch on or off computer instances on their cluster. The user can install any operating system and run on those machines any service, also allowing the execution of computing intensive tasks.

In the remaining of this chapter we will present in detail the available solutions for the execution of Bag-of-Tasks problems, with respect to available infrastructures, tools, and programming methodologies. Next we present the deficiencies of the available solutions, when applied to our focus populations, and list the objectives and contributions of or work.

1.1 User classes

Up until a few years ago the only users that could take advantage of parallel computing infrastructures where those performing numerical simulations. These users where required to have knowledge of parallel programming, as they had to develop the jobs completely: processing code, task creation, and communication between tasks.

Today, new classes of users requiring fast processing of data emerged. Some actually have some programming language (but not necessarily with proficiency on parallel programming) but others don't. These only know how to work with a limited set of tools, mostly by invoking them with command line arguments

Here we present the current existing class of users that may require high performance or parallel computing infrastructures:

- Expert HPC users (**with** parallel programming knowledge)
- Programmers (**without** specific parallel programming knowledge)
- Tool users and Hobbyists

Expert HPC users are researchers requiring large infrastructures to execute their simulations. The codes developed simulate physical phenomenon, namely on the physics, chemistry or mechanical engineering area. The developed code runs on hundreds of nodes and, due to their nature, require communication among nodes. These users, when developing their simulation code, have to take into account the inter-process communication and the synchronization between processes. Some of these users have access to institutional Clusters on the Grid.

Another class of users knows how to program and can develop sequential applications. However, they can only develop, often sophisticated, single threaded simulation code, that nowadays can be executed efficiently on personal computers. Some of the simulations require the execution of the same code with different parameters. These users can be found in research laboratories working in the area of biology, or even computer science (e.g. network protocols simulation).

Nowadays, users of proprietary tools (such as image rendering software, or data analysis) can also be a target population for parallel execution environments. These users know how to parametrize a limited set of tools, but at some points of their work need to execute repetitive tasks over a large set of data. These users usually have access to personal computers, and their computer knowledge is often limited. Hobbyists also fit in this category of users, in case of processing batches of images or rendering images.

1.2 Current tasks execution environments

Today, the sources for computing cycles are diverse and more close to the common user present in the edges of the Internet. In this section we will address these sources (from the classics to the most recent ones) and evaluate their target population and possibility to be used by a common user.

These sources range in terms of availability and computing power:

- Institutional HPC Clusters
- Grid
- Personal Clusters

- Internet Distributed computing systems,
- Utility computing

High Performance Computing (HPC) infrastructures are composes of hundreds of computing nodes and several TByte of storage space, all connected with dedicated high speed network links. This allows the fast execution any demanding computing jobs: Bag-of-Tasks, master-slave or HPC applications with interprocess communication. These systems are usually owned and managed by an entity the enforces strict access and usages polices: only user belongings to that organization are allowed run jobs, or the access requires a previous contract. So, the access to these systems are restricted to users with a continuous and high demand for computing cycles.

Grid infrastructures ease the remote access to HPC computing infrastructures to users outside the owner organization (e.g. in the context of a virtual organization). Furthermore Grid initiatives and infrastructures allow the aggregation of scattered resources. In the same manner as with classical HPC Cluster it is necessary for a user to use it to have a institutional relationship with a grid initiative participant.

The construction of a small scale computing cluster is nowadays easy. The available commodity off the shelf (COTS) hardware is powerful enough to be used in complex computations, and the speeds attained with a simple Ethernet gigabit switch are sufficient to the user needs. Although the hardware is available, the existing tools are not target at all the user classes. Intensive computing applications may also take advantage of multicore machines, but the integrated execution of multiple distributed computers is not easy to the common user.

The processing power of desktop personal computers has increased in last years. Adding to this fact, these personal computers, besides being idle a large part of the time, have increasingly faster Internet network connections. Taking advantage of these facts a new infrastructure for parallel computing has emerged: Internet Distributed computing. Nowadays several research projects and organizations exist that take advantage of this novel computing environment: researchers develop applications that perform a certain simulation to be executed on the personal computers owned by the the users in the edge of the Internet. Data is transferred to the donor computer when the user is connected to the Internet allowing later execution of the simulation (when, otherwise, the computer would be idle).

Recently private data center owners with a presence in the Internet have begun to offer computing services to remote users, making the emergence of the Cloud. One of such services (namely Amazon Elastic Cloud) allows computer instances (virtual machines) to be rented for the execution of off-the-self operating systems. Such services allow users to install ordinary versions of regular operating

systems, and on demand launch instances of those operating systems. The charged value depends on execution time and on the hardware characteristics (number of processors, memory). The access to these computer instances is straightforward, as the user only has to sign a simple contract and provide a valid credit card to be later charged.

1.3 Current parallel programming paradigms

Depending on the problems being solved different programming paradigms and work organization and decomposition fit them better. In this section we present the main used programming paradigms for the development of parallel jobs and the various ways to organize the tasks comprising a job.

1.3.1 Work organization

The kind of problem being solved determines the best suited work organization to be used, being one of the following:

- Bag-of-Tasks
- Master-slave
- Recursive
- Decomposition with inter-task communication

Bag-of-Tasks problems are composed of independent tasks. These tasks can be launched independently of each other. Before execution, each computing node receives the code and the data to be processed and, after the execution of the code, the results are returned. At the end all tasks execution, the user has a set of data that must be later processed. In this kind of jobs there is no interaction between the launching code and each task, furthermore after each task execution (the processing of data) the launching code (part of the job) does not further process the results.

In a master slave problem, there is some interaction between the main process (master) and the tasks (slaves), but still there is not any communication among tasks. The master, besides launching each task, also performs some data pre-processing and results aggregation. In the simplest form, the master only interacts with the slaves at the beginning of each task (to invoke and send data) and when tasks finish, to retrieve results. More complex interactions involve the master to wait for the slaves (using barriers) in the middle of their execution, in order to retrieve partial results, process them and distribute new versions of data.

If tasks can spawn themselves in order to further distribute work, we are before a recursive work distribution. The data assigned to a task is split and part of it is assigned to the newly created child

task. A task can only terminate (and return its result) after completion of all its children and aggregation of their partial results.

The most complex problems require data to be decomposed and distributed among processes and, during execution, communication between the various tasks. When, in order to a task to make progress in its computation, it needs data from another tasks (i.e. a sibling or neighbor task, w.r.t data being processed), communication between computing nodes needs to be carried out. When developing tasks' code, it is necessary to guarantee the correct synchronization so that dead locks do not happen. Although a master-slave solution is always possible, the number of tasks, and the amount of transmitted data may render a centralized solution unusable. The decentralizes communication between tasks eliminates a bottleneck (the master) and reduces communication delays.

1.3.2 Programming model

In order to implement a project and its tasks some sort of programming tools and libraries must be used. These libraries or tools must allow the implementation and deployment of the previously describe work organization, follow some known model, and fit into at least one of the following classes:

- Message passing communication
- Shared memory communication
- Explicit programmed task creation
- Declarative task definition

of the previous list the two fist item state how communication between components is programmed and carried out. The other two define how tasks are created by the programmer/end user.

When the work organization requires communication between components (master-slave or decomposition with inter-task communication), the programmer either uses an explicit Message Passing API or use of shared memory paradigm.

With message passing, the programmer explicitly invokes functions to send and receive data. During execution, the executing processes (tasks or master) synchronize at these communication points, with the receiver of the message waiting for the sender to transmit the data. In a master-slave problem, usually the master waits for messages from all the tasks, processes the received data and transmits back a new set of data for a new parallel interaction. In there is inter-task communication, tasks must know the identification of those others where some data must be sent to, to periodically

communicate with them.

Another way to transmit information among tasks or between a master and its slaves is using variables residing in a shared memory space. These variables are shared among the participant processes, and its access is performed as if it was a simple global variable, by using the common programming language assignments, accesses and expressions. As this data can be concurrently accessed by various processes, it is necessary to guard these accesses: synchronization routines must be explicitly used when reading or writing values on the shared memory.

Shared memory is mostly used when executing the concurrent tasks in a shared memory multiprocessor, as the available memory space is naturally shared among all the process or threads. This solution incurs no performance penalty when performing communication, because there is no data transmission between private memory spaces. When the tasks are to be executed on a cluster of computers connected by a System Area Network (SAN) (such as Gigabit ethernet or Myrinet), a Distributed Shared Memory [34] (DSM) library can be used. Although not all the data can or should be shared among tasks, a communication buffer can be shared and accessed as a local variable. In order to reduce data transfer these DSM systems implement data coherence protocols that guarantee that, from all the shared data, only the necessary one (the one recently modified) is transmitted. This data selection is performed without the user knowledge or intervention and transparently reduces the overall communication [56, 58].

In terms of task creation, the user either creates them explicitly (in inter-task communication, master-slave or recursive problems) or relies on the infrastructure to create them. Using a task creation API, the programmer must define when tasks are to be created: in the beginning of the program or during the jobs execution.

If the problem fits into the Bag-of-Tasks category, it is only necessary for the programmer to define the code to be executed by each task. The underlying system will be responsible for launching the tasks (transmitting the code and the input data of each task) and retrieve the partial results after each task completion. The code to be executed can be enclosed in either a function (or class method) or a self-contained application. In both scenarios the underlaying system is responsible for the execution of that code.

1.4 Objectives and contributions

In the following tables, we systemize the relations between each class of users and the infrastructures, tools and methodologies previously presented.

Table 1.1 shows the types of computational work the different classes of users usually have to perform.

	Experts	Programmers	Tool users
Bag-of-Tasks	✓	✓	✓
Master-slave	✓	✓	✗
Recursive	✓	✗	✗
Inter-task communication	✓	✗	✗

Table 1.1: Usual work organization by user class

As explained earlier **expert users** have the knowledge and the need to solve any kind of problems, from the simplest **Bag-of-Tasks** to those with **inter-task communication**, where exists communication between individual computing nodes.

Both **programmers** and **tool users** can develop solutions to problems fitting the **Bag-of-Tasks**. They can develop serial applications (or use pre-existing tools) that execute in a single processor. Some of their problems can be solved with the repetition of the execution of such serial applications. In the case of programmers it is possible for them to implement simple **master-slave** problems.

The following table (Table 1.2) presents the possibility for a particular user to access and use the defined computing resources.

	Experts	Programmers	Tool users
HPC Clusters	✓	✗	✗
Grid	✓	✗	✗
Personal Cluster	✓	✓	✓
Distributed Computing	✓	✓	✗
Utility Computing	✓	✓	✓

Table 1.2: Ease of access to the various infrastructures

As expected, the **expert users** have access to any of the presented infrastructures. If the institution he is working at has a Cluster or belongs to one of the existing Grid initiatives, the access to those resources is possible. The other two classes of users, because they do not have the necessary institutional links, are deprived of these resources.

Today any kind of user, independently of his programming skills can build, with off the shelf components (computing and network hardware), a small **personal cluster**. This cluster can also aggregate old and slower hardware. Although small, these can speed some of the usually lengthy jobs.

The access to utility computing infrastructures (such as Amazon Elastic Cloud [2]) is also possible to any of the presented user classes, as it is only necessary to sign a simple contract with the service provided

Distributed Computing may seem a solution usable by any kind of users, but that is actually not the case. Any user with a personal computer and an Internet connection can donate cycles for solving others problems, but not everybody can take advantage of them. Today it is necessary to have programming knowledge to write the code to be remotely executed. Furthermore it is necessary to be well known and to have some media coverage to gather enough donors.

Another relevant issue is how well a user is proficient in a required tool to execute their parallel jobs, Table 1.3 shows this relation.

	Experts	Programmers	Tool users
Message Passing	✓	✗	✗
Shared Memory	✓	✗	✗
Task creation	✓	✓	✗
Task definition	✓	✓	✓

Table 1.3: Ease of use of various programming models and tools

As expected **expert users** are able and can use any kind of tool or API to develop their jobs.

Users with limited programming skill may not be able to use **message passing** (such as MPI) or **shared memory** APIs to develop their jobs, but are able to design and develop self contained tasks that are created using a simple **task creation** API or using a **task definition** tools. **Message passing** or **shared memory** requires an architectural knowledge that a non expert programmer may not have.

On the other end of the spectrum we have **tool users** that can not use any system requiring programming knowledge. These users are limited to the declarative **task definition**.

1.4.1 Objectives

Observing the previous tables we can conclude that although **expert users** can take advantage of the full spectrum of systems, work organizations and resources, this is not true to the other two classes of users.

Programmers with limited parallel programming knowledge can only solve **Bag-of-Tasks** and simple **master-slave** problems, execute their jobs on **personal clusters**, **Distributed computing** infrastructures and **Utility computing** systems, and develop them using simple **task creation** APIs or **task definition** tools.

Ordinary **tool users** are further limited: the resources are limited to **personal clusters** and **utility computing**, and they can only develop their **Bag-of-Tasks** jobs using **task definition** tools.

Besides these natural restrictions, these two user classes face further limitations due to the inadequacy of the available tools to these new users and usage.

Although it is possible to easily build personal clusters, the tools to deploy work on them are still the same as the ones available to HPC infrastructures, making them impractical to low knowledge users. With utility computing the same happens: it is possible to create on-demand clusters but the tools to easily deploy work on them do not exist. Furthermore, nowadays it is impossible to know exactly how many machines should be allocated for the intended performance, so that the execution cost is minimal.

The Internet Distributed computing may seem the optimal source for resources, but users with short term jobs, or with small visibility, can not take advantage of this cycles source. On the following chapter (Chapter 2), we present a taxonomy for Internet based Distributed Computing, and apply it to existing systems in order to know the recipes for success of such systems, and why these systems are not in more widespread use.

Our work focuses on solutions for the execution of Bag-of-Tasks problems created by users with limited or no knowledge on parallel programing and without access to conventional parallel execution environments. We try to solve some of the limitations of present environments. We explore three specific lines of work addressing different aspects, ranging from high level task creation to lower level resource discovery:

- novel task creation methods
- new parallel execution environments
- new resource integration and discovery methods

We can match these lines of work to different layers of the overall architecture presented in figure 1.1. The following paragraphs present briefly the developed work, that will be substantiated in Chapter 3.

Jobs submission Although the Bag-of-Tasks are the easiest problems to develop, the existing tools are not the best suited to our target population.

In our work, we developed mechanisms targeting efficiency (in terms of development time) and expressiveness in task creation, allowing the submission of jobs and their tasks based on independent applications [61] or independent method invocations [62].

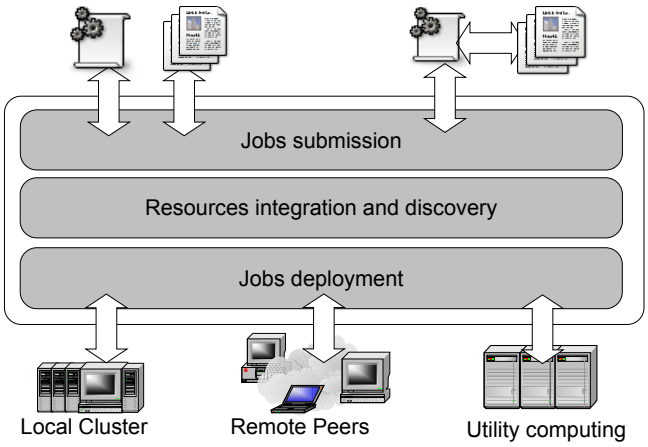

Figure 1.1: Simplified architecture

Jobs deployment The job deployment mechanisms and easy of use the available execution environments are also important for the less knowledgeable user.

We addressed two new execution environments: Internet distributed computing environments [60] and utility computing infrastructures [59, 63] . Furthermore part of our work allows easier employment of available personal clusters [61].

Resources integration and discovery Without efficient mechanisms for resource discovery, aggregation, integration and usage, the previously presented work is of no use.

We developed an heuristic for efficient use of an Utility computing infrastructures as a source of processors for the execution of Bag-of-Tasks [59, 63], and developed mechanisms and algebras for a more precise evaluation of Internet scattered resources [57].

1.5 Document Roadmap

The rest of the document is organized as follows. In the next chapter we present an extract of a taxonomy developed , and use it to discover the requirements for a successful system: usable by users of all conditions, and providing efficient and correct results.

In the Chapter 3 we present the overall architecture of developed work and its various components, refering the reader to the specific papers for the whole contents.

1.6 Scientific Publications

All the developed work was presented in the context of peer reviewed international scientific conferences and workshops, and partially described in a book chapter. Currently, an article submitted for

publication on a scientific journal is also under review. The list of these publications is as follows:

- A2HA - Automatic and Adaptive Host Allocation in Utility Computing for Bag-of-Tasks. [63]
  João Nuno Silva, Luís Veiga, Paulo Ferreira.
  Submited for publication on JISA - Journal of Internet Services and Applications, Springer.
- Peer4Peer: E-science Communities for Overlay Network and Grid Computing Research. [67]
  Luís Veiga, **João Nuno Silva**, João Coelho Garcia. (2011).
  Chapter to appear on the book "Guide to e-Science: Next Generation Scientific Research and Discovery", Springer.
  ISBN: 0857294385
- Service and resource discovery in cycle-sharing environments with a utility algebra. [57]
  João Nuno Silva, Paulo Ferreira and Luís Veiga. (2010).
  In 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS). IEEE.
  Object Identifier: 10.1109/IPDPS.2010.5470410
- Mercury: a reflective middleware for automatic parallelization of Bags-of-Tasks. [62]
  João Nuno Silva, Luís Veiga, and Paulo Ferreira. 2009.
  In Proceedings of the 8th International Workshop on Adaptive and Reflective MIddleware (ARM '09). ACM.
  Object Identifier: 10.1145/1658185.1658186.
- SPADE: scheduler for parallel and distributed execution from mobile devices. [61]
  João Nuno Silva, Luís Veiga, and Paulo Ferreira. 2008.
  In Proceedings of the 6th international workshop on Middleware for pervasive and ad-hoc computing (MPAC '08). ACM.
  Object Identifier: 10.1145/1462789.1462794
- Heuristic for resources allocation on utility computing infrastructures. [59]
  João Nuno Silva, Luís Veiga, and Paulo Ferreira. 2008.
  In Proceedings of the 6th international workshop on Middleware for grid computing (MGC '08). ACM.
  Object Identifier: 10.1145/1462704.1462713
- nuBOINC: BOINC Extensions for Community Cycle Sharing. [60]
  João Nuno Silva, Luís Veiga, and Paulo Ferreira. 2008
  In Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops. IEEE.
  Object Identifier: 10.1109/SASOW.2008.66

2 Internet based Distributed Computing

The use of a Distributed computing infrastructure may seem the panacea for common users with computational needs. The true fact is that there are several systems developed to be user by users scattered in the Internet, but in fact are not.

In this chapter we briefly present a new taxonomy for the characteristics of Distributed Computing Systems. For each aspect, we first highlight the possible alternative aproaches and leave further details for later analysis in the Section. Then we conclude with a critical evaluation of the reasons why these systems are not widely used and present the case of BOINC with its solutions to increase the user participation.

This taxonomy includes the more usual architectural characteristics but also those more tied with the user experience and often overlooked: efficiency of job execution, security and mechanisms for development and creation of jobs. Using the presented taxonomy, we will also characterize the most relevant systems developed up to date.

A more detail presentation of the various alternatives for the characteristics presented is available on a separate document [65].

2.1 A Taxonomy for Cycle-Sharing Systems

Three important factors impact the user experience one can have while using a Distributed Computing System, either when submitting work or executing it: i) the architecture, ii) the reliability, and iii) how the user interacts with the systems.

The first class of factors limits the overall performance of the systems when the number of users (providing and using resources) scales to hundreds, or more, and how efficiently the executing hosts are selected (taking into account the jobs requirements). Delays before the start of each task should be minimum and fairness should be guaranteed when tasks from different users compete for the same resource.

The security is an important feature, as the resource providers do not want to become vulnerable to attacks and those that have work to be done require some degree of privacy and correctness guarantees.

The last class affects how the user creates jobs and what kind of jobs can be submitted. A system

difficult to configure will not attract donors nor the clients will be able to efficiently create their jobs.

As all these factors directly affect the user (as a work creator or donor), they are fundamental to guarantee that a system gathers a large user base to be useful to the clients.

The remaining of this section is split in three parts, each for one of the classes of characteristics. There, the different characteristics are presented and then applied to the various Distributed Computing Systems considered in this study.

The main focus of this document are systems that are public in the sense that they are generic enough to allow any Internet connected computer owner to create projects or jobs, and that o not require complex administrative donors admission. Such systems are generic, not being tied to a specific problem, and should allow users to create jobs (by providing the processing code and data) or just create tasks (by submitting the data to be processed by previously developed and/or deployed code).

In our target systems, users can be either clients or donors. Donors are gathered from the Internet, not requiring them to belong to an organization (as seen in Enterprise Desktop Grids). These donor users are only required to install some simple software module that will execute the code submitted by the clients. The client users have problems to be solved requiring complex easily parallelizable computations.

The presented systems allow clients to solve computational problems by deploying them on the Internet to be executed by donors. The problems are solved by the execution of a job. These computational jobs are composed of the execution code and data to be processed. In Bag-of-Task or Embarrassingly parallel problems the jobs are composed of multiple independent tasks. All tasks execute the same code, but process different data. As each tasks is independent from the others, they can be executed concurrently in several donor computers. Jobs can also be aggregated in projects.

The use of non dedicated network distributed machines for the execution of parallel jobs has been initiated with Condor [35]. This infrastructure allows the execution of independent tasks on remote computers scattered on a LAN. As the original target computing environment was composed of commodity workstations, Condor only scheduled work when these computers were idle. Due to the homogeneity of the environment (all aggregated workstations shared a similar architecture and operating system) compiled applications could be directly used.

The applicability of the concept inaugurated by Condor to the Internet was limited by the natural heterogeneity of this new environment. There was no guarantee that the available workstations shared the same architecture, or operating systems, and no widely available portable language existed. The development of the JAVA VM and language language solved these problems.

This new language was portable to most existing architectures, allowing the execution of a single application version on distinct devices. This allowed the development of the first generic Distributed Computing System. The second half of the nineties witnessed an increased development rate, with the application of novel techniques to solve the presented problems. We offer an exemplificative yearly distribution of the work:

- **1995** : ATLAS [14]
- **1996** : ATLAS [9] and ParaWeb [15]
- **1997** : IceT [27], SuperWeb [52], JET [48, 64], Javelin [19, 16] and Charlotte/KnittingFactory [10],
- **1998** : Java Market [4, 3], POPCORN [45] and Charlotte/KnittingFactory [29, 11],
- **1999** : Javelin [43], Bayanihan [51, 50] and Charlotte/KnittingFactory [12]
- **2000** : Javelin [44]
- **2001** : Bayanihan [50]

In the XXI century the development and research on Distributed Computing continued, seeing the advent of the Peer-to-Peer architectures for resources discovery and work distribution:

- **2000** : MoBiDiCK [20] and XtremWeb[26]
- **2001** : XtremWeb[23]
- **2002** : JXTA-JNGI [69, 42], P^3 [46] and CX [17]
- **2003** : G2-P2P [38, 39]
- **2004** : CCOF [70]
- **2005** : Personal Power Plant [55], CompuP2P [54], G2DGA [13], Alchemi [37, 36] and BOINC [7]
- **2006** : YA [18] CompuP2P [28], BOINC [6]
- **2007** : Leiden[66, 33], Ginger/NuBoinc [68] BOINC [5]
- **2008** : Ginger/NuBoinc [60]
- **2010** : Ginger/NuBoinc [49, 22]

In a similar way as with Distributed Computing systems, work has also been done in the aggregation of both institutional clusters and donors to the grid.

For instance, in Albatross [8, 30] most work has been done in the development of an infrastructure for the consolidated use of distributed clusters. Albatross optimizes work distribution taking into account communication latency between clusters. LiveWN [31] allows ordinary users to donate cycles to a grid, by executing grid middleware inside a virtual machine. Although users can easily donate cycles, a pre-existent grid infrastructure (with all complex security configurations) must exist.

These systems are of no interest to this study since some sort of prior organized administered infrastructure should exist (clusters in Albatross and a Grid in LiveWN), making them impractical, or otherwise inaccessible and/or unusable to a regular home user.

2.1.1 Architecture

Although the CPU speed of the donating hosts is the most important factor to individual task execution time, the various architectural decisions have a fundamental impact on the overall system performance, such as jobs speedups, fairness on the selection of tasks, improvement or optimization of resources utilization.

With respect to architectural characteristics, we present different implemented network topology and organizations, how resource are evaluated, what scheduling policies are used and how work distribution is performed.

2.1.1.1 Network Topology

Different systems offer different network topologies, with different organization of the various involved entities (donors, clients, or servers).

This architectural decision affects the kind of entities, their interaction, the overall complexity and the system efficiency. These can be, in order of increasing flexibility and decoupling:

- Point-to-point
- Single server
- Directory server
- Hierarchical servers
- Replicated servers
- Peer-to-peer

2.1.1.2 Resource Evaluation

Tasks require various kinds of resources to be efficiently executed. Due to the heterogeneity of the donors it is necessary to keep track of the characteristics of the donors to optimize tasks execution.

The way available resources characteristics are monitored, evaluated, and information about them is kept is also an important factor for the overall system performance. There are three different approaches:

- Centralized database

- Decentralized database
- Polling

2.1.1.3 Scheduling Policies

Along with the resource discovery and evaluation, the way jobs are scheduled is also important to the performance of the system. The available policies range from the simpler one (eager) to a complete heuristic matchmaking between available resources and tasks requirements:

- Eager
- Resource aware
- Heuristic
- Market oriented
- User priority

2.1.1.4 Work Distribution

Both the network architecture and scheduling policies affect the way work distribution (from servers to donors) is performed. Work distribution can be characterized in two independent axis: i) who initiates the process, and ii) whether it is brokered by a third party:

- Direct pull
- Direct push
- Brokered pull
- Brokered push

2.1.1.5 Analysis

Table 2.1 systematize the various combinations of architectural characteristics of the available Internet Distributed Computing systems described in this section.

	Network Topology	Resource Evaluation	Scheduling Policies	Work Distribution
ATLAS	Hierarchical servers	-	Eager	Direct pull
ParaWeb	Multiple servers Directory server	Polling	-	Direct push
Charlotte	Single server	Centralized DB	Eager	Direct pull
KnittingFactory	Directory server	-	-	Brokered pull
SuperWeb	Single server	Centralized DB	-	Direct push

Table 2.1 (Continues on next page)

	Network Topology	Resource Evaluation	Scheduling Policies	Work Distribution
Ice T	-	-	-	-
JET	Hierarquical servers	-	Eager	Direct pull
Javelin	Hierarquical servers	Decentralized DB	Eager	Brokered pull
Java Market	Single server	Centralized DB	Market oriented	-
popcorn	Single server	Centralized DB	Market oriented	-
Bayanihan	Hierarchical servers	Decentralized DB	-	-
MoBiDiCK	Single server	Centralized DB	Resource aware	-
XtremWeb	Single server	Centralized DB	Resource aware	Brokered pull
JXTA-JNGI	Replicated servers	Decentralized DB	Eager	Brokered pull
P^3	Peer to Peer	Decentralized DB	-	Brokered pull
CX	Multiple servers	Decentralized DB	Eager	Direct pull
G2-P2	Peer to Peer	Polling	Eager	-
CCOF	Peer to Peer	Decentralized DB	Resource aware	Brokered push
Personal Power Plant	Peer to Peer	Polling	Eager	Direct pull
CompuP2P	Peer to Peer	Polling	Market oriented	Brokered push
Alchemi	Single server	Polling	Eager	-
YA	Peer to Peer	Decentralized DB	Resource aware	Brokered push
BOINC	Point to Point	Centralized DB	Eager Resource aware	Direct pull
Leiden	Point to Point	Centralized DB	Eager	Direct pull
Ginger	Peer to Peer	Decentralized DB	Resource Aware	Brokered pull
NuBoinc	Point to Point	Centralized DB	Eager	Direct pull

Table 2.1: Architectural decisions of Internet Distributed Computing systems

Network Topology The late nineties saw the development of many systems, each one with a different approach to its architecture. While on some systems the architecture was not a fundamental issue (using simply point-to-point or a single server), for others the architecture was fundamental (due to efficiency issues and due to the proposed programming model).

The simpler point-to-point architecture was initially implemented by Para-Web, while the use of a different server from the client computer was implemented by Charlotte, SuperWeb, Java Market, POPCORN and MoBiDiCK.

In order to balance load across servers, JET uses a two layer hierarchical architecture, where there is a single JET server and a layer of JET Masters, that interact with the donors. Bayanihan uses a similar approach to tackle network limitations, taking advantage of communication parallelism and locality of data.

In Atlas and Javelin any task information is stored in one of the available servers. Each one of these servers also manages a set of clients. Furthermore, as will be presented in section 2.1.3.10, these systems allow any tasks to create and launch new ones.

These two characteristics allow a simple load balancing mechanism. Whenever a new task is created, its data is stored in the least loaded server. As previously executing tasks depend on new ones it is necessary to maintain connections between the several servers hosting related (child and parents) tasks, creating a tree based structure.

Furthermore, both ATLAS and Javelin also allow work stealing as a mean to distribute work. Whenever a donor is free to do some work, it contacts a server. If that server has available work it assigns it immediately to a donor. In the opposite case the initially contacted server finds a server with available work. This way a hierarchical and recursive architecture also emerges.

Most of later projects present a Peer-to-Peer architecture, where donors also act as servers and clients with management tasks (storage of work and results, distribution of work, ...): XtremWeb, P^3, G2-P2, CCOF, Personal Power Plant, CompuP2P and YA.

JXTA-JNGI clearly makes a distinction between donors, clients and servers but uses a Peer-to-Peer infrastructure to manage communication between replicated servers.

Ginger implements a peer to peer network topology, with distinction between regular and Super Nodes. These special nodes store information about the reputation of a set of regular nodes.

Other recent projects implement simpler architectures: CX, Alchemy, BOINC and Leiden. CX allows the existence of multiple servers (each one managing distinct sets of work) and Alchemi that only uses a single server. BOINC and Leiden and nuBOINC rely on the simple point-to-point architecture for the distribution of work.

Resource Evaluation The systems that require up-to-date knowledge of the donors characteristics perform polling whenever work is sent. In the case of G2-P2, Personal Power Plant and CompuP2P, whose architecture is peer-to-peer, it is natural that polling is required to know the exact characteristics of the donors. In the case of ParaWeb and Alchemi, no information about donors is stored in the server so it is necessary to poll them whenever work is to be executed.

All other systems, independently of the architecture use some sort of database. If the architecture relies on multiple servers (hierarchical, replicated or peer-to-peer) the database is decentralized.

Scheduling Policies The simplest of the scheduling policies (Eager) is also the most adopted. This is due to the use of an interpreted language (as will be seen in Section 2.1.3.7), and its homogeneous execution environments. Although systems have different execution power (available memory and execution speed), on systems using Eager scheduling these characteristics are not taken into account when selecting the host to run a task.

The systems that are resource aware assign tasks taking into account donors' characteristics (processor, operating systems, ...) or availability. For instance MoBiDiCK donors specify the time slot where their computers can donate cycles, this information is used when scheduling tasks. CCOF uses the same idea of optimal resource availability and automates the execution and migration of tasks. Here, users do not have to specify the availability slots, as CCOF uses current time and assumes

computers are only available during night. Tasks are scheduled taking into account donors' current time, selecting donors that are available and also imposing the migration of executing tasks at dawn. In XtremWeb, BOINC and Leiden clients must develop one executable for every architecture, so it is necessary to match the donor architecture (processor and operating system) with the suitable executable. Ginger goes one step forward, as the matching of clients with donors also takes into account the reputation, and historic data, of the donors.

Java Market, CompuP2P and POPCORN use a market oriented approach, by matching a value offered by the client with the one required by the donor. In Java Market this matching is performed automatically, after the client submitting the required resources and the value he is willing to give back for the execution of the task in certain amount of time. In the case of concurrent execution of tasks, the ones that maximize benefits for the donors are chosen. CompuP2P uses Vickrey auctions to assign tasks to the less expensive donor, after donors stating the cost of the resources. POPCORN offers three different auction types: Vickrey, a sealed-bid double auction (where both parties define a lower and higher bound for the price of the resource) and a repeated clearinghouse double auction.

Work Distribution The way the work distribution is performed is partly dependent on the system architecture. Systems that rely on a single server necessarily perform direct task distribution, either pulling or pushing work.

In systems with more complex architectures the distribution can be brokered so that the donor does not have to know and contact directly the server or peer owning the task (KnittingFactory, Javelin, XtremWeb, JXTA-JNGI, P^3, CCOF CompuP2P YA and Ginger). In other systems the various servers (or peers) are used to find the interlocutor, but task transfer is performed directly between the donor and the server that stores the task information (JET, CX and Personal Power Plant).

The work distribution method (pull or push) is also related to the scheduling policies. Systems that have an eager scheduling policy use a pull mechanism: when idle, the donor contacts a server or another peer and receives the work to be processed. The distribution can be both direct or indirect, as previously explained.

2.1.2 Security and reliability

The second class of relevant characteristics of Distributed Computing systems are those related to security and reliability, either on the donor side and on the client side: i) privacy of the data, code and client identity, ii) result integrity against malicious donors, iii) Reliability against donor and network faillure, and iv) security guarantees of the donor computer.

2.1.2.1 Privacy

Some of the work that can be deploy and executed on the Internet can have sensitive information: the data being processed or even the code to be executed. The identity of the user submitting the work should, in some cases, be kept secret.

So, on the client side, these Distributed Computing systems should guarantee the following kinds of privacy:

- Code
- Data
- Anonymity

2.1.2.2 Result Integrity

As most machines that execute tasks on Distributed Computing systems are out of control of the user submitting jobs, it is fundamental to verify result correctness after the conclusion of a task. Only if that happens it is possible to guarantee that those results are the same as the ones that would be obtained in a controlled and trusted environment.

There are several techniques to guarantee that the results are not tampered nor forged:

- Executable verification
- Spot-checking
- Redundancy
- Reputation

2.1.2.3 Reliability

Even if donors are not malicious and execute all tasks correctly, they can always fail or get disconnected from the Internet. It is still necessary to guarantee that in case of failure, the job gets executed and results produced. This can be accomplished using one of the following techniques:

- Redundancy
- Restarting
- Checkpointing

2.1.2.4 Execution Host Security

On the side of the donor some security precautions should be taken into account in order to prevent malicious code to execute and cause any harm. To avoid such problems a few solutions are possible:

- User trust
- Executable inspection
- Sandboxing

2.1.2.5 Analysis

Next table presents how each system solves the security and reliability issues.

	Privacy	Result Integrity	Reliability	Execution Host Security
ATLAS	Anonymity	-	Checkpointing	User trust Sandboxing
ParaWeb	-	-	-	-
Charlotte KnittingFactory	-	-	Redundancy	Sandboxing
SuperWeb	-	-	-	Sandboxing
Ice T	-	-	-	User trust Sandboxing
JET	-	-	Checkpointing	Sandboxing
Javelin	-	-	Redundancy	Sandboxing
Java Market	-	-	-	Sandboxing
popcorn	-	-	-	Sandboxing
Bayanihan		Redundancy Spot-checking	Redundancy	Sandboxing
MoBiDiCK	-	-	Redundancy	-
XtremWeb	-	Executable verification	Restarting	User trust
JXTA-JNGI	-	-	-	-
P^3	-	-	Checkpointing	-
CX	-	-	Redundancy	-
G2-P2	-	-	Checkpointing	-
CCOF	-	Spot-checking Reputation	-	Sandboxing
Personal Power Plant		Redundancy Voting	-	-
CompuP2P	-	-	Checkpointing	-
Alchemi	-	-	-	Sandboxing
YA	-	-	-	-
BOINC	-	Redundancy Reputation	Restarting Checkpointing	User trust
Leiden	-	Redundancy Reputation	-	User trust
Ginger	-	Spot-checking Reputation	-	Sandboxing
nuBOINC	-	Redundancy Reputation	-	User trust

Table 2.2: Security concerns

Privacy In terms of privacy, only anonymity can be guaranteed by available systems. Privacy of the code and data is never referred in the literature and difficult to guarantee. Although some systems use virtual machines (mostly Java or .Net virtual machines) to execute the code, after the data and code have been downloaded, malicious processes running on the donor can access that information.

Only the papers describing ATLAS refer client anonymity, in this system a donor can not obtain

the identity of the client that submitted the work. Although not explicitly stated with respect to other systems, any system that does not have a point-to-point architecture (with distinctions between server and client computers) can easily guarantee that anonymity.

Other systems (BOINC, ParaWeb, Charlotte/KnittingFactory, POPCORN) rely on the knowledge of the identity of the client. In these systems the donor explicitly chooses the jobs and projects he allows to run on his computers.

Result Integrity Most of the studied systems do not present any solution to guarantee result integrity on the presence of malfunctional or malicious donors.

From those that take into account result corruption, redundancy is the solution adopted by the majority (Bayanihan, Personal Power Plant, BOINC, Leiden and nuBOINC). These systems launch several identical tasks and, after receiving the results, decide about the correct answer. Most of the systems resort to a vote counting mechanism on the server side: a result is considered valid if a majority of computers returned that value. Personal Power Plant, due to its peer-to-peer architecture, must use a distributed voting algorithm.

Bayanihan, CCOF and Ginger uses spot-checking to verify the correctness and trustiness of a donor, issuing dummy tasks with a previously known result.

XtremeWeb donor software calculates the checksum of the downloaded executable before starting the tasks. By comparing it with the checksum calculated on the server it is possible to confirm that there was no executable tampering. Besides executable verification, XtremeWeb offers no other mechanism to verify if the results transmitted from the donor were the ones calculated by the verified executable.

CCOF, BOINC, Leinden, Ginger and nuBOINC implement reputation mechanism to guarantee that non trustable donors do not interfere with the normal activity of the system. CCOF resorts to the results of spot-checking while BOINC, Leiden and nuBOINC use previously submitted results to classify donors in different trust levels. Ginger uses results from spot-checking and real tasks to calculate the reputation of a donor.

Reliability Reliability, guaranteeing that a task eventually completes, is not tackled by some of the studied systems.

As expected, CompuP2P (a system that is market oriented) implements checkpointing. For a market oriented system any other method would increase the value to pay for the execution of a task: redundancy requires the launch of several similar tasks, even if there is no need, and with task restarting the work done until the donor failure would be wasted. For the other market oriented systems

(Java Market and POPCORN), there is no information about the mechanisms to assure some level of reliability. Five more systems implement checkpointing with recovery of the tasks on a different donor: ATLAS, JET, P^3, G2-P2 and BOINC, with different levels of abstraction.

G2-P2 periodically saves the state of the application and records all posterior messages. These checkpoints can either be saved on the donor local disk or on another peer. The first method is more efficient, but does not guarantee that a task can be recovered.

Checkpointing is the only method that can be used with tasks that have side effects. The other methods, which are also the simplest require all tasks to be both independent and idempotent. Three systems implement the restarting of tasks when only some results are missing to complete a job: XtremWeb, BOINC and Leiden.

In system where resources are truly free, redundancy can be used without further cost to the client. Although redundancy can be used to guarantee result integrity and system reliability, there are some systems that do not take advantage of redundancy to tackle both issues. As a mean to guarantee result integrity, BOINC only uses redundancy when requested by the user, while Charlotte, Javelin, Bayanihan, MoBiDiCK and CX do not perform any result verification.

Execution Host Security Although systems that use an interpreted language to implement tasks execute the code within a virtual machine, not all of these are capable of guaranteeing donor host integrity against malicious code. In the case of systems that use Java, only those that have tasks implemented as applets (ATLAS, Charlotte, SuperWeb, IceT, JET, Javelin, Java Market, POPCORN, Bayanihan, CCOF) execute the code inside a sandbox. Ginger also executes its tasks inside a virtual machine. Alchemi uses the .NET virtual machine that allows the definition of sandboxes.

In ATLAS, IceT, XtremWeb, BOINC, Leiden and nuBOINC donors simply trust the developers of the tasks, believing that the code will not harm the computer.

2.1.3 User interaction

The way users interact with the available Distributed Computing Systems may also be relevant to the popularity of such systems. We present the roles a regular Internet user can have in such systems, the mechanisms to create jobs and tasks, and the incentives for donating cycles for the community.

2.1.3.1 Work Organization

Independently of the terminology used by each author, the work submitted to a Distributed Computing System follows a predetermined pattern, with a common set of entities and corresponding

layers.

The work can be organized around the following entities, of increasing complexity:

- Task
- Job
- Project

Systems organizing work in different layers use more complex entities, not only to encase other simpler entities, but also to reduce the effort to create work.

2.1.3.2 User Roles

A regular user located on the edge of the Internet can have several roles in a Distributed Computing System:

- Donor
- Job creator
- Project creator

It is obvious that any user can install any of the available systems and be capable of creating and submitting work to be executed.

In this classification we make the distinction between the administrator of the computer hosting the work and the user submitting it. If it is necessary for a user to be the administrator (because of the inexistence of user level tools) to create jobs, we do not consider possible for a regular user to create them.

Some systems allow a single user to have several of the presented roles.

2.1.3.3 Job Creation Model

A job creation is always composed of two steps: definition of the processing code, and definition of each task input parameters and data. In systems with projects these two steps are independent.

The processing code definition can be performed in three different ways. The user can take advantage of off-the-shelf applications or develop his own processing code:

- (COTS) Executable assignment
- Module development
- Application development

The way the job code is defined or developed affects the way tasks are created, this relationship will be further described later in the analysis.

2.1.3.4 Task Creation Model

To create a job it is necessary to define the various independent tasks that will compose it. Depending on the system, the development and definition of those tasks can be made in different ways:

- Data definition
- Data partition
- Code partition

2.1.3.5 Task Execution Model

During execution of the project there are different ways to start and execute the various tasks, leading to different organization of these tasks:

- Bag-of-tasks
- Master-slave
- Recursive

2.1.3.6 Offered Services

Although the programmer must always develop the code that is to be executed on the remote hosts, the available systems can provide different support tools or services. For the definition and deployment of parallel jobs, the offered services can fit into:

- Launching Infrastructure
- Monitor Infrastructure
- Programming API

Some sort of software layer (middleware) must always exist in order to coordinate all available resources, independently of the offered support services. The infrastructure referred in this section does not deal with this concern but with the launching and definition of jobs and tasks.

2.1.3.7 Programming Language

The programming language used for the development of projects and tasks, not only limits the type of problems one can solve, but also affects how easily development is carried out.

In the different phases of the development and execution of tasks several kind of languages, can be used:

- Declarative

- Compiled
- Interpreted
- Graphical

Some of the presented systems used different languages for the development of projects and definition of tasks.

2.1.3.8 Project Selection

Some of the available systems can host several projects, so there is a need for donors to select the projects they want to donate cycles to. Independently of the mechanisms to select the projects, the selection process fits into one of these classes:

- Explicit
- Restricted (topic based)
- Implicit

2.1.3.9 Donation Incentives

To gather donors, each system must give back some reward for the donated time. Available systems, after the completion of task assigned the donor, may reward the donor with one of the following:

- User ranking
- Processing time
- Currency

In any of the presented incentives, for the complete execution of a task some reward is awarded. This reward can be just a point/credit, the right to use another CPU, or a virtual monetary value (currency) to use in other services.

In either of the three cases, the reward should take into account the processing power employed in the execution of each task. In these scenarios, the processing time is not a good measure, as a slow machine takes longer to process a task.

2.1.3.10 Analysis

The way the various systems implement and handle the user interaction issues is presented in two distinct tables: Table 2.3 (specifying the various possible user roles) and Table 2.4 (presenting programming and usage alternatives).

	Work organization	User Roles	Job creation Model	Task creation Model	Task execution Model
ATLAS	Job Task	Donor Job creator	Application dev.	Code partition	Recursive
ParaWeb	Job Task	Job creator	Application dev.	Code partition	Bag-of-tasks
Charlotte KnittingFactory	Job Task	Donor Job creator	Application dev.	Code partition	Master-slave
SuperWeb	Task	Donor Task creator	Module dev.	Data definition	Bag-of-tasks
Ice T	Job Tasks	Donor Job creator	Application dev.	Code partition	Master-slave
JET	Job Task	Donor Job creator	Module dev.	Data partition	Bag-of-tasks
Javelin	Job Task	donor Project creator	Application dev.	Code partition	Recursive
Java Market	Task	Donor Task creator	-	Data definition	Bag-of-tasks
popcorn	Job Task	Donor Job creator	Application dev.	Code partition	Bag-of-tasks
Bayanihan	Project Job tasks	Donor Project creator Job creator	Application dev.	Data partition	Master-slave
MoBiDiCK	Project Job Task	Donor	Application dev.	Data partition	Bag-of-tasks
XtremWeb	Project Job Task	Donor	Application dev.	Data partition	Bag-of-tasks
JXTA-JNGI	Project Job Task	Job creator Donor	Application dev.	Code partition	Master-slave
P^3	Job Task	donor Job creator	Module dev.	Data partition	Bag-of-tasks
CX	Job Task	Donor Job creator	Application dev.	Code partition	Recursive
G2-P2	Job Task	Donor Job creator	Application dev.	Code partition	Master-slave
CCOF	-	-	-	-	Bag-of-tasks
Personal Power Plant	Job Task	Donor Job creator	Application dev.	Code partition	Master-slave
CompuP2P	Job Task	Donor Job creator	Executable def.	Data partition	Bag-of-tasks
Alchemi	Job Task	Donor Job creator	Application dev. Executable def.	Code partition Data definition	Master-slave Bag-of-tasks
YA	-	-	-		
BOINC	Project Job Task	Donor	Application dev. Executable def.	Data partition	Bag-of-tasks
Leiden	Task	Donor Task creator	Application dev.	Data definition	Bag-of-tasks
Ginger	Job Task	Donor Job creator	Executable def.	Data definition	Bag-of-tasks
BOINC	Job Task	Donor Job creator	Executable def.	Data partition	Bag-of-tasks

Table 2.3: User roles

Work Organization Most of the system evaluated divide the work in both jobs and tasks. In these systems the work submission unit is a job composed of tasks with similar code but different input

data.

SuperWeb, Java Market, and Leiden do not have the job concept. User can only submit individual tasks that have no relation between them.

On the other hand, Bayanihan, MoBiDiCK, XtremWeb, JXTA-JNGI and BOINC have the project entity. Before any job creation, it is necessary to define a project. The responsibility for the project creation differs in these systems, as we will see later.

User Roles As expected, all systems allow users scattered on the Internet to donate their cycles on a simple, and some times anonymous, way. What distinguishes most systems is the ability of ordinary users (those that can donate cycles) to create work requests (tasks, jobs) and make them available to be executed.

On BOINC, XtremeWeb and MoBiDiCK only the administrator can create work. On all other systems, the creation of jobs (and tasks) is straightforward without requiring any special privileges.

Job Creation Model and Task Creation Model The way jobs and tasks are created is tightly linked in Distributed Computing Systems.

In three systems, the user that has work to be done has to explicitly assign the data to each task: SuperWeb, Java Market and Leiden. These are also the systems that do not have the concept of job, here every task is truly independent of the others. In SuperWeb the programmer develops a module whose code will be executed by each task, while in Leiden, the tasks correspond to a independent application that is executed on the remote host.

We can observe that the other systems (JET and P^3) that require the sole development of a module (the code of the tasks) only require data partitioning when creating tasks. Although the user is required to develop the code to split the data, no explicit creation of tasks is necessary.

In MoBiDiCK, XtremWeb, CompuP2P, Alchemi and BOINC each task is encased in a complete application that executes on the remote hosts. This application must specially be developed. In Ginger and nuBOINC the user also assigns a executable to the job, but does not have to implemente it. These applications are commodity off the self one that are widely available (or installed) on the donor computers. As in these systems, each task corresponds to the execution of a complete application, the creation of tasks is simply made by assigning to each one its input data. In these systems, with the exception of Alchemi, the user must program the data partition. In Alchemi by means of XML file the user defines each task input data. In BOINC and Alchemi it is also possible to use as task code a pre-existing application (close to the concept present on nuBOINC and Ginger).

In Bayanihan the user develops the main application that is responsible for data partition. In this system, tasks are not created explicitly. The data to be processed is programatically stored in a pool, by means of a supplied API. The system will then pick data from that pool and assign it to the tasks, without programmer intervention.

In all other systems the developed application must contain, as a function or class, each task's code. The data partitioning and task creation are performed internally in the developed application.

Task Execution Model With the exception of Bayanihan, in all other systems whose tasks are created by means of data Partitioning, the parallel jobs fall in the Bag-of-Tasks category. In these systems it is impossible to have some sort of workflow (where results are used as input of other tasks) and there can only be some minimal pre-processing of the data and post-processing of the results.

Although in Bayanihan tasks are created by data partition, the programmer can both control and interact with the running tasks and chain them to get complex data computations and workflows.

In the two systems that require the explicit declarative definition of data (Java Market and Leiden), the solvable problems obviously fall in the Bag-of-Tasks category.

All other systems can be used to solve master-slave problems, and consequently also Bags-of-Tasks.

ATLAS, Javelin, and CX also allow the execution of recursive problems, by allowing tasks to spawn themselves to create new tasks.

	Offered Services	Programming Language	Project Selection	Donation Incentives
ATLAS	Programming API	Interpreted	Implicit	-
ParaWeb	Programming API	Interpreted	Implicit	-
Charlotte KnittingFactory	Programming API	Interpreted	Explicit	-
SuperWeb	Infrastructure	Interpreted	Implicit	Processing time
Ice T	Programming API	Interpreted	-	-
JET	Monitor infrastructure Programing API	Interpreted	-	-
Javelin	Launch infrastructure Programming API	Interpreted	Implicit	-
Java Market	Infrastructure	Interpreted	-	Currency
popcorn	Programming API	Interpreted	Explicit	Currency
Bayanihan	Monitor infrastructure	Interpreted	Implicit	-
MoBiDiCK	Launch infrastructure Programming API	Compiled	Explicit	-
XtremWeb	Infrastructure	Interpreted	Implicit	-
JXTA-JNGI	Programming API	Interpreted	Implicit	-
P^3	Programming API	Interpreted	-	-
CX	Programming API	Interpreted	Implicit	-
G2-P2	Programming API	Interpreted	Implicit	-
CCOF	-	-	-	-

Table 2.4 (Continues on next page)

	Offered Services	Programming Language	Project Selection	Donation incentives
Personal Power Plant	Launch infrastructure Programming API	Interpreted	Explicit	-
CompuP2P	Infrastructure	Interpreted Declarative	Explicit	Currency
Alchemi	Infrastructure Programming API	Interpreted Declarative	Implicit	-
YA	-	-	Implicit	
BOINC	Monitor infrastructure Programming API	Compiled	Explicit	User ranking
Leiden	Infrastructure	Declarative	Explicit	User ranking
Ginger	Infrastructure	Declarative	Explicit	Currency User ranking
nuBOINC	Infrastructure	Declarative	Explicit	User ranking

Table 2.4: Programming and usage

Offered Services Some of the analyzed systems only provide a programming API for the development and launching of applications. Programmers write one application, launch it and wait for the resulting tasks to finish. In such systems there is no way for the owner of the work to control the various tasks: i) terminate the tasks, ii) check for their status, or iii) observe the intermediate results.

SuperWeb, Java Market, XtremeWeb, CompuP2P, Alchemi and Leiden, on the other hand, offer a full infrastructure for the deployment and launching of jobs and for observing tasks execution status.

Other systems also have some sort of infrastructure either for launching jobs (Javelin, MoBiDiCK and Personal Power Plant, Ginger and nuBOINC) or for monitoring tasks (JET, Bayanihan and BOINC).

Of the available systems, some require the programmer to use a programming API to develop the application, and use the infrastructure to launch the jobs or monitor the task. Examples of such systems are JET and BOINC (with a programming API and monitoring infrastructure), and Javelin, MoBiDiCK and Personal Power Plant (with a programming API and launching infrastructure).

Programming Language The large majority of the available systems use interpreted languages for the development and execution of tasks, namely Java. MoBiDiCK and BOINC require the development of a compiled application.

In CompuP2P, Alchemi, Leiden, Ginger and nuBOINC the creation of tasks resorts to the declaration of their arguments. In Alchemi the user defines them in a XML file, while in the other systems (CompuP2P, Leiden, Ginger and nuBOINC) the user uses a supplied user interface.

Project Selection In most systems the user has no way to select which projects he will donate cycles to. In these systems it is the server that selects what tasks to send to the clients, and as each server hosts several projects the donor does not know where the task belongs to.

In systems where a server only hosts one project, the project selection is obviously explicit. The donor contacts a pre-determined server, knowing exactly what is the purpose of the tasks being run.

In BOINC the donor contacts one particular server, but can select the projects that he wants to donate cycle to. From the various hosted projects the server selects tasks from the ones the user has previously registered.

In Ginger and nuBOINC the selection is also explicit as the donor selects what off the self application can be user to execute remote tasks.

Donation Incentives Most of the studied systems do not have any sort of reward to users donating cycles. In the market oriented systems (Java Market, POPCORN and CompuP2P), the reward for executing correctly a task is a currency value that can later be used to get work done remotely. In SuperWeb instead of using a generic currency the donor is rewarded a certain amount of processing time to be spent later. In Ginger the user is rewarded a generic currency, that can be later be used to exchange for processing time, but his reputation (user ranking in the table) is also modified. The overall user ranking is used when assigning work to donors.

The donors in BOINC, Leiden and nuBOINC are only rewarded execution points, that serve no other purpose than to rank the donors. These points can not be exchanged for work and are only used for sorting users and the groups they belong to.

2.2 Evaluation

We can point a few characteristics that have a greater impact on user adhesion to systems and the donation of cycles to other users. A system that optimizes (by using the most efficient technique) these characteristics is most capable of gathering donors. These characteristics fall under the previously presented classes:

- **Security** - Execution host security
- **Architecture** - Network topology and scheduling policies
- **User interaction** - User roles, project selection and donation incentives

One of the fundamental issues that may prevent users from donating cycles to others is the security of their machines. There should be some sort of guarantee that the downloaded code will not harm the donor machine. The use of some sort of sandbox (either application or system level) is the best approach to guarantee this. By isolating the downloaded code, even if it is malicious, the host computer can not be compromised.

Of the proposed Network Topologies, Peer-to-peer is the one that more easily allows users to adhere to a system, continue using it and donate cycles to others. In a Peer-to-peer infrastructure, no complex configurations are needed. The peer configuration (usually stating a network access point) is much simpler than configuring a client to connect do different servers. After this initial configuration the donor automatically is in position to donate cycles to any client, making resources (in our case cycles) more available than on client server systems.

Besides simplicity of configuration, current peer-to-peer systems (mostly on file sharing) offer other highly appreciated characteristics. In these systems anonymity is usually preserved, but still allowing the aggregation of users around common interests. The preservation of these characteristics would also increase user adhesion.

Another factor that can increase user participation (by donating cycles), is the possibility for users to take advantage of the system. If users are able to execute their work (having the role of job creators) they are more willing to donate cycles later. Furthermore, there should be some fairness on the access to the available cycles. The selection of tasks to be executed must have into account the amount of work the task owner has donated to others. This issue can be handled by the scheduling algorithm when selecting the tasks to be executed.

The use of a market oriented approach can also introduce a level of fairness. Users receive a payment for the execution of work and use that amount to pay for the execution of their own tasks.

Necessary for fair task execution are the rewards given for the execution of work. If, when scheduling tasks, user sorting or market mechanisms is used, it is necessary to use some differentiator values (either points or some currency). These reward points or currency are given after each task completion and can later be used to sort the tasks or for biding resources.

If users know the available projects, and what problems are being solved by the tasks, they are more inclined to donate cycles. By knowing what the tasks do, users can create some sort of empathy, and donate cycles to that particular project. To guarantee this, it is necessary that job selection is explicit.

Taking into account the more relevant characteristics one may be tempted to think that the most used Distributed Computing platform (BOINC [53]) implements the majority of the best solutions to each problem.

In the following list we present how BOINC implements the most relevant characteristics:

- **Execution host security** - User trust
- **Network Topology** - Point-to-point
- **Scheduling Policies** - Eager / Resource aware

- **User Roles** - Donor
- **Project Selection** - Explicit
- **Donation Incentives** - User ranking

The first two issues, those more related with the system architecture, present sub-optimal solutions. There is no automatic mechanism to guarantee the security of the donor, and the network topology may not efficiently handle a large amount of users.

The use of a single server for a project is justified by the fact that a single computer can handle more requests than those possible in a real execution case [7]. The absence of any security guarantee mechanism is much more difficult to justify. Users just trust the code they are downloading from a server, because they know who developed it and what is the purpose of the work being executed. This is only possible because project selection is explicit.

In BOINC, users donating cycles can not submit their own work. Their sole function is to execute tasks created by the project managers. With such usage, the scheduling policies are not relevant to the satisfaction of the users. To promote cycle donation BOINC employs user ranking: one of the rewards is to be seen as the better donor, the one executing more tasks.

The two fundamental decisions for the success of BOINC as a infrastructure for Distributed Computing are: the explicit selection of projects and user rankings.

The explicit selection of projects allows user to donate cycles to those projects they think are more useful, thus leveraging the altruistic feelings users may have. As most projects have results with great impact to society, e.g. medicine, it is easy for them to gather donors and become successful.

On the other hand, the ranking of users based on the work executed promotes competitive instincts. Although donating cycles to useful causes, the ranking (of users and teams) increases the computing power a user is willing to donate to a cause.

These two factors greatly overcome the deficiencies of BOINC (security, and user roles) and make BOINC arguably the most successful Distributed Computing system.

2.3 Summary

In this chapter we briefly presented a taxonomy for the evaluation of Internet based Distributed Computing systems. We presented the main characteristics (architectural, security and reliability and related with the user interaction) and described how each available system implement them.

We also performed a critical evaluation of the relevant characteristics influential to the success of such systems, and evaluated the most successful system (BOINC).

3 Architecture for new Execution Environments

The Figure 1.1, presented in Chapter 1, can be further detailed showing how the developed work fits in the overall architecture presented. The developed components and the interaction among them can be seen in further detail in Figure 3.1.

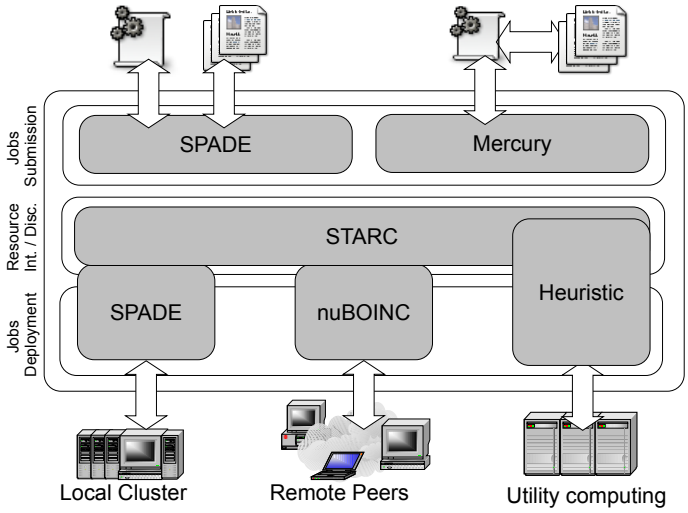

Figure 3.1: Architecture

The work was evenly distributed among the three relevant layers presented:

- Jobs submission - the top layer responsible for handling user requests for job submission and creation;
- Resource Integration and discovery - the core layer responsible for the efficient use of all the available resources, independently of their source, and invocation interfaces;
- Job deployment - the layer that interacts with the resource instances selected.

With respect to job submission, the available systems studied (such as Condor [35], Ganga [21]) and programing methodologies (threads, message Passing API [40, 25] , or even Aspects) do not offer solutions to the new user classes with low (or even none) programming knowledge. So we defined a user interface for Bag-of-tasks' creation (**SPADE**) and a platform for automatic code parallelization (**Mercury**), whose results were presented in International Scientific Conferences and published in [61, 62], and are further described in Sections 3.4 and 3.2

We also envisioned new environments for the execution of Bag-of-tasks, namely the Cloud and the edge of the Internet. In **nuBOINC** [60], we present an architecture for an off-the-shelf Distributed

Computing system that allows any user to submit work and execute other users' tasks.

Our work on job deployment on the Cloud (module **Heuristic** in Figure 3.1) deals with two different aspects: the execution of independent tasks in a Utility Computing infrastructures, and the efficient allocation of hosts that participate on the computation.

In order to integrate the various available computing sources, an efficient evaluation mechanism is essential, to do so we designed and developed **STARC** [57], an algebra and an extensible architecture for precise evaluation of resources.

Each one of the remaining sections of this chapters describes the previously presented modules, also stating their specific goals and results, and this theses' overall contributions.

3.1 Resource Discovery Algebra

The possible use of Internet scattered resources by new classes of users poses a new challenge on the evaluation of the required resources. Besides new users providing new resources (with new, different and more diverse characteristics) these users also have different requirements

Today, the cycles usable by users may be harvested from several scenarios, ranging from college or office LANs, cluster, grid and utility or cloud computing infrastructures, to peer-to-peer cycle sharing overlay networks. The user requirements are also different, besides the usual requirement pair $< number\ of\ processors, memory >$ users may require more complex resources such as a specific language interpreter, library, or tool.

Adding to these changing environments , we now have different user classes, requiring each one a differentiated offered quality of service: users with a institutional link to the resource providers should be offered all the resources required, while unregistered users (using donated resources) can be satisfied with only a partial fulfillment of their requirements.

Existing resource discovery protocols have a number of shortcomings for the existing variety of cycle sharing scenarios. They either i) were designed to return only a binary answer stating whether a remote computer fulfills the requirements, ii) rely on centralized schedulers (or coherently replicated) that are impractical in certain environments such as peer-to-peer computing, or iii) they are not extensible as it is impossible to define new types of resources to be discovered and evaluated or new ways to evaluate them.

In order to solve current shortcomings, we developed a novel, extensible, expressive, and flexible requirement specification algebra and resource discovery middleware. Besides standard resources (memory, network bandwidth,...), application developers may define new resource requirements

and new ways to evaluate them. Application programmers can write complex requirements (that evaluate several resources) using usual logic operators. These complex requirements are evaluated in different ways, depending on the class of the user, as presented in the next sub-sections.

3.1.1 Contributions

This work is divided in four different aspects:

- Requirement description language
- Individual resource evaluation
- Complex requirement evaluation
- User classes

The developed requirement language follows an hierarchical organization whose DTD is presented in Figure 3.1.

```
1 <!ELEMENT requirement (resource | and | or | not )>
2 <!ELEMENT and         (requirement+)>
3 <!ELEMENT or          (requirement+)>
4 <!ELEMENT not         (requirement)>
5 <!ELEMENT resource    (config+)>
6 <!ELEMENT config      (#PCDATA)>
7 <!ATTLIST resource name CDATA #REQUIRED>
8 <!ATTLIST requirement policy (userclass | priority| strict | balanced|elastic) "userclass">
9 <!ATTLIST requirement weight CDATA "1.0">
```

Listing 3.1: XML requirement DTD

The outermost structures are the usual logical operators (**and**, **or** or **not**) that can be nested along with the leaf **resource** element.

In the definition of a specific resource (line 7) the user specifies, along with its name, a snippet of XML that is to be evaluated by the resource provider.

Each individual requirement is evaluated in the resource provider. A novel feature presented in our work is the run-time download of a code module responsible for the evaluation of the available resource. This feature allows the extensibility of the evaluation systems with the definition (by the administrator or end-user) of new resources to be evaluated.

Another innovative feature is the use of fuzzy values and operators on the evaluation of resources and complex requirements. An evaluation returns a real value between 0.0 and 1.0 stating the fulfillment level: i) 0.0 means the requirement is not satisfied, ii) 1.0 means the requirements is fully satisfied, and iii) a value in between represents the level of fulfillment.

The blind calculation of the returned value will render a straight line 0.0 and 1.0 depending on the level of partial fulfillment of the requirement. This may not be satisfactory, as the user levels of satisfiability for a variation of the characteristics of a resource, may not be linear. To solve this problem we allow the user to define a intermediate utility depreciation function: besides stating the preferred resource (evaluated of 1.0), the user defines intermediate utility values for known characteristics (pairs $< characteristic, utility\ values >$, where the utility value varies between 0.0 and 1.0). The evaluation of a resource, and its utility function, will no longer be linear, but composed of straight segments, that join the absence of the resource (returning 0.0) to the complete fulfillment (returning 0.0), passing by the user defined utility values.

The evaluation of the complex requirements (**and**, **or** or **not**) uses different operators depending on the classes the user belongs to.

Policy	Intended users	Calculation	AND (aggregation)	OR (adaptation)	NOT (disapproval)
priority	administrators	lexicographic	must all occur	follow priority list	reject and fail
strict	SLA users	Zadeh fuzzy logic	min	max	complement (1-x)
balanced	registered members	geometric average	product	max	inverse (1/x)
elastic	best-effort	arithmetic average	sum (averaged)	max	opposite (-x)

Table 3.1: Combined Evaluation Policies of Aggregate Utility in STARC

The defined user classes range from the **administrators** (whose requirements must be completely satisfied) to **best-effort** users that can be partially satisfied. This admitted satisfiability level is attained using different logical operators classes:

- Boolean logic operators
- Zadeh logic operators
- Other fuzzy logic operators

For each user a single logic operator class is used, allowing the values returned for the evaluation of different resources to be used to sort them.

3.1.2 Implementation Highlights

The requirement description language was described in a DTD (presented in Listing 3.1) and a parser for it was developed. As a prof of concept, we developed STARC, a middleware capable of run-time loading (from a local or remote repository) and executing of the resource evaluation code.

The XML requirements parser and the requirement evaluation middleware was developed in Python, thus allowing an easy integration of downloaded code execution.

Although a myriad of resources are possible to be evaluated by the STARC system (due to the easy development and integration of resource evaluation code), we developed a series of modules for the evaluation of the more usual resources (such as memory, number of processors) in the most usual computer architecture and operating systems (Microsoft windows, Linux, and Mac OS X).

3.1.3 Results

STARC is capable of evaluating and comparing different resource providers with respect to client specific resource requirements. This evaluation return a continuos but not necessarily linear utility value of a specific resource. The values return from the evaluation of a resource are between 0.0 and 1,0, but the user can assign specific utility value to precise partial fulfillments of the requirements.

We were able to evaluate a set of standard resources in different computer architectures: number and speed of CPU cores, available memory, available disk space, network speed, among others, according to different client profiles.

With the proposed utility algebra and corresponding XML DTD it is possible to define any kind of requirement a module or a complete application may have and that a resource provider must satisfy. The use of various logic operators (namely fuzzy logic operators) eases the comparison of different hosts. By using these operators the result of the evaluation returns a numeric value that clearly states how the requirements are totally or partially fulfilled. The values returned by all hosts can be easily compared to find the one(s) that best fit the client requests, w.r.t. all required resources and perceived utility depreciation (partial-utility), according to a combined evaluation policy. This way, resource discovery is more effective (and will result in fewer resource discovery failures) than a simple matching approach.

The architecture of STARC allows its extensibility by allowing the definition of new types of resources to be discovered and evaluated. The code to evaluate a resource can be dynamically installed, without system compilation, and reused on behalf of many clients.

3.2 Automatic Parallelization

Nowadays there are two ways to create Bag-of-Tasks, i.e. embarrassingly parallel application: i) the user either develops a complete application and uses a middleware to invoke the various instances of it (one for each individual task), or ii) uses an API to create different threads on a local or remote host.

If the user is required to develop an application and create inside of it the various concurrent tasks, the available APIs were not designed for this kind of problem. For instance, MPI allows the parallel execution of tasks, but was developed for much complex parallel applications, with high data communication between tasks. The use of such APIs requires the programmers to learn them, and add complexity to the final parallel solution.

Even if a simple API exists, the user has to learn how to use it, and the application becomes tied to it, leading to difficult, even impossible, porting to different execution platforms.

We developed Mercury, a software layer that provides a platform for the transformation of serial applications (similar in structure to the example presented in Listing 3.2) into parallel Bag-of-Tasks. Mercury reads an external declarative configuration file stating what methods and classes should be parallelized, loads the application, and in run-time transforms it so that the specified methods are executed concurrently. This transformation is performed without further user intervention. Its modular design allows the integration of Mercury with different parallel environments.

```
1 for i in range(1000):
2   inputData = getTaskInput(i)
3   objList[i] = processinObject()
4   objList[i].processData(inputData)
5 for i in  range(1000):
6   outputResult = objList[i].getResult()
7   process(outputResult)
```

Listing 3.2: Tipical serial Bag-of-Tasks version

3.2.1 Contribution

While application transformation can be performed off-line, either at compilation time or as a intermediate step, this requires the user to know beforehand if the application is to be transformed or not.

Our solution proposes the use of run-time application adaptation: when executed, Mercury checks for the necessity of, or advantage in, transforming the application (for instance, if several processors are available) and transforms the application.

After the transformation, the methods that were previously defined as concurrent are executed in different threads. The code to be parallelized must fit the pattern presented in Listing 3.2.

The user must only state in a configuration file the name of the method to be executed in a different thread, in this example it is the method `processData`, and ensure that **Mercury** is installed.

When loading the application, **Mercury** inserts the necessary code to launch each `processData` method invocation on a different threads, and sets up a barrier. Only after completion of all the `processData` methods, the main thread may continue executing the `getResult` methods.

3.2.2 Implementation Highlights

Mercury was developed in Python, thus allowing the parallelization of Python written applications. Mercury main component is a metaclass, that intercepts every application loading and manages every class instantiation and object creation.

This metaclass intercepts every regular class loading, checks if it has any method to be executed concurrently and, if so, injects modified constructors and synchronization methods.

After being loaded, an application contains, beside the original class, one transformed class and adaptors to the various existing parallel execution environments, as shown in Figure 3.2.

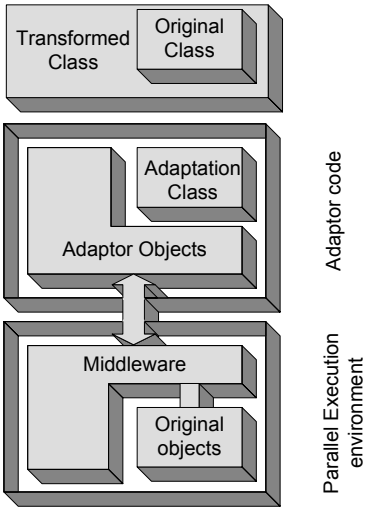

Figure 3.2: Transformed classes organization

The unmodified code interacts transparently with a **transformed class** (when creating objects) or with instances of this class. The **transformed class** handles every **original class** object creation method, while its instances intercept all calls to the instances of the **original class**.

The instances of the **transformed classes** contain the synchronization code and interact with the parallel execution environments (such as local or remote threads), by means of the **adapator objects**.

3.2.3 Results

With Mercury we manage to automatically and transparently produce parallel Bag-of-Tasks. Any user, with a minimum of programming skills, only has to develop a serial version of its job (following simple restrictions), define what objects and methods should be executed concurrently and execute that application within Mercury.

Mercury transparently handles the parallel, and possibly distributed, execution of the lengthy methods taking advantage of available processors (in a multicore machine or on a cluster).

The initial experiments done show that the overhead is minimal, and that it is possible to take advantage of parallel processing environments (multiprocessors/multicores, clusters, ...) without the use of complex APIs.

3.3 Off the shelf Distributed Computing

The use of the computers scattered on edge of the Internet as a source of computing may seem a good solution to the execution of Bag-of-tasks. This is true to projects with high visibility, as proved by the success of BOINC [47], but to the common user that is not true. To these users, cycle sharing over the Internet is a one-way deal.

Computer owners only have one role in the process: to donate their computers' idle time. This is due to the fact that it is difficult for an ordinary user to install the required infrastructure, develop the processing applications and gather enough computer cycle donors.

With nuBOINC we developed a set of BOINC extensions that allow any user to create and submit jobs that can take advantage of remote idle cycles. These jobs are processed by commonly available software (e.g. programming language interpreters or virtual machines, statistical software) that is installed in the remote donating computers.

In order to submit their jobs, users only have to provide the input files, select the processing application and define the command line to provide to that application. Later, users of the same software packages will contact the server, receive a set of jobs, and process them using the already installed commodity application. These users can later take advantage of other people's computer cycles.

This system allows an expressive definition of jobs providing considerable speed gains, while leveraging a cycle-sharing platform and widely available commodity applications, in a truly global communal computer cycle market.

3.3.1 Contributions

nuBOINC tries to explore some of BOINC's characteristics that lead it to be the most successfully Distributed Computing infrastructure to date, while at the same time eliminating some barriers to an easy deployment.

We propose the use of commodity applications (such as data processing applications, image rendering engines, image manipulation applications) as the tasks execution environments. Users will use these applications to solve their problems, but at a later time will feel compelled to donate cycles to users using the same applications. With nuBOINC users know what is the purpose of the donated cycles and can have some level of interest and attachment to the problems being solved.

By providing an easy to use task definition user interface, the difficulty of creating parallel jobs is reduced, as the user only has to supply the data (multiple files, or different arguments) for each task. The use of these widely available applications greatly augments the available donor base.

3.3.2 Implementation Highlights

The original BOINC infrastructure was used as a development base, only being modified to accommodate the new user role: jobs creator. The modifications are as following:

- Client side application registrar
- Job submission user interface
- Modified BOINC client
- New database tables
- Modified BOINC server scheduler

The two first modifications deal directly with the new role users have: clients that submit jobs.

The **Application registrar** is necessary in order to allow the user to define what commodity off--the-shelf applications can be used to execute others' jobs. This information is stored locally on the client and used when executing tasks, and stored in the server to be used when scheduling them.

The **Job submission user interface** (Figure 3.3) replaces the internal mechanisms to create jobs, allowing any user, even without programming knowledge, to submit work to be executed.

The user just states the application to run the tasks, and its command line arguments. The distinction between tasks is made by means of a placeholder (e.g. %(ID)02d) to be used when defining the command line arguments.

The **modified BOINC client** is executed on the donor computer and when receiving a nuBOINC task invokes the registered application with the data received from the server.

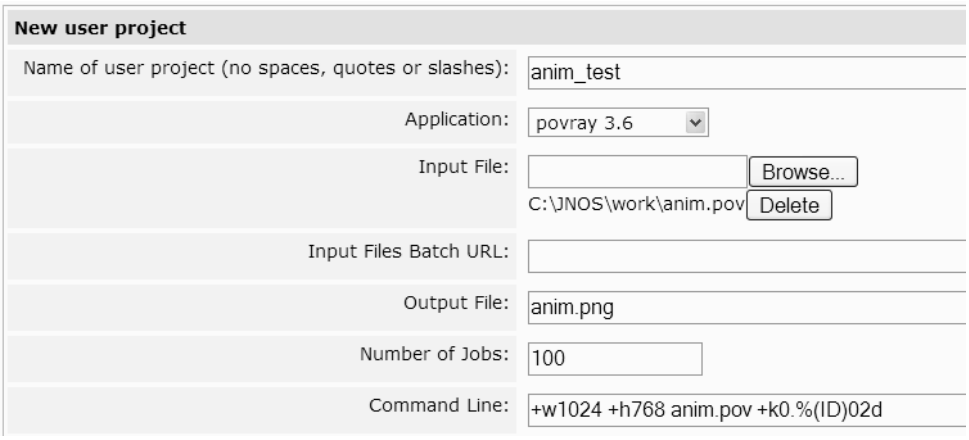

Figure 3.3: User project submission interface

On the server side, the modifications deal essentially with data storage and task scheduling. New data (registered applications and links between users and their jobs) need to be stored, and the scheduling of the jobs must take into account the installed and registered application on the client computer.

3.3.3 Results

The developed work allows the definition of some of the existing Bag-of-task problems capable of being solved with off-the-shelf applications (image/video rendering, image processing, and even execution of complete interpreted applications).

The possible speedups depend on the number of donor computers, but easily gains can be obtained. If the client has more than one available computer, nuBOINC not only provides an easy to use jobs definition infrastructures, but allows immediate gains from remote execution of the job on multiple computers.

Security is a major concern and accounting is a fundamental functionally that are out of scope of this work.

3.4 User interface for task creation

The most easy to develop jobs are those whose tasks are composed of a complete application to be executed with a set of input files and command line arguments.

The existing middleware for deployment of independent tasks over remote computers (such as Condor [35] or Globus [24]) require users to write complex launch scripts and configuration files and to know their syntax.

Graphical tools (such and Ganga [21] and Nimrod [1]) allow the definition of jobs, but are not suitable to the average user. Ganga does not allow the splitting of a job in several tasks, while Nimrod, requires the existence of a previously defined template.

To users just wanting to use their idle personal computers, or wishing to use a system such as nuBOINC, none of the available solutions are usable.

The development of user interface for job submission targeted at users without large computing experience, requires a previous study of the possible problems to be solved. These fit, obviously, into the Bag-of-Tasks problems, where the differences between each tasks fall into one of the following:

- each task processes a different input file;
- each tasks receives different values as arguments;
- each task processes a different part of a (uni or bidimensional) set.

This classes of work division are observed in most jobs.

The batch processing of photos or data file analysis requires each task to get as input a file with a different name. The user should be able to easily state how many tasks to create and what file is to be processed by each task.

If that what distinguishes tasks is a numerical scalar argument, the user should define the tasks arguments based in the task identifier, either directly or by means of a simple transformation function.

Jobs such as image rendering, that can be split assigning to each task, a part of a bidimensional area, are also usual. In this case the user should only be required to state the limits of the set and how many tasks to create.

3.4.1 Contribution

We developed SPADE, a tasks scheduling middleware, target as mobile devices, but requiring an easy to use job definition interface. In order to allow the definition of the previously described kind of jobs, we developed an expressive user interface (shown in Figure 3.4), where the user states how the data (input files, numerical values, os sets) are to be split among tasks.

The fundamental concepts on this job definition method are the placeholders. Placeholders are used in the definition of the input, and output file names, and in the argument definition, and are later replaced by a value at task creation time.

The essential placeholders are as follows:

- **%(ID)d**

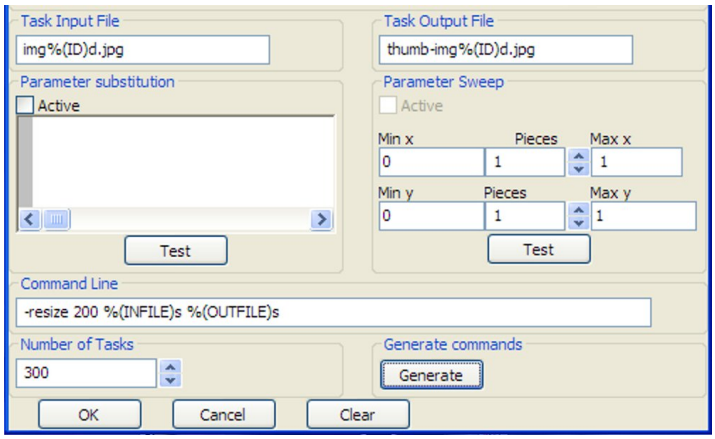

Figure 3.4: Job submission user interface close-up

- **%(OUTFILE)s**
- **%(INFILE)s**

One of these placeholders, **%(ID)d**, is replaced by the tasks identifier (a value between zero and the number of tasks). This placeholder is to be used in the definition of the input/output files names and in the definition of the command line.

In the definition of the command line the user should also define the arguments based on the placeholder **%(INFILE)s** and **%(OUTFILE)s**.

By also defining the processing application (not shown in Figure 3.4) and the number of tasks, simple tasks can be created.

For the definition of more complex data splitting a few more placeholders were defined:

- **%(NEWPARAM)s**
- **%(MINX)s** and **%(MAXX)s**
- **%(MINY)s** and **%(MAXY)s**

If the application arguments or input/output files do not depend directly on the tasks identifiers, it is possible for the user to write in Python a simple transformation function that, for each task assigns the **%(NEWPARAM)s** placeholder a different value.

For the splitting of a contiguous set of values (uni or bidimensional) the user must fill the corresponding fields in the **Parameter Sweep** area. At their creation, each task is assigned a different set of placeholders **%(MINX)s**, **%(MAXX)s**, **%(MINY)s** and **%(MAXY)s**) that defines the limits of the area processed by that tasks.

3.4.2 Implementation Highlights

This user interface was implemented in python and integrated with a simple task scheduling middleware (SPADE) targeted at mobile devices.

The implemented runtime allows the definition of task parameters (command line arguments and input/output file) for the most usual jobs splitting classes described earlier.

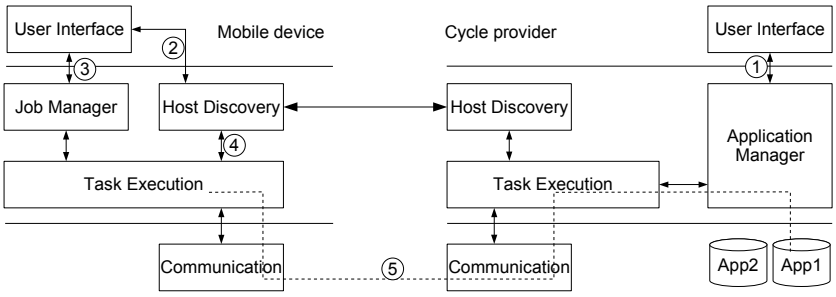

Figure 3.5: SPADE System Architecture

The overall architecture where our job definition user interface was integrated is presented in Figure 3.5. Although the the developed infrastructure was targeted at mobile computing environment, this was work can easily be ported to classical cluster or even utility computing infrastructures.

3.4.3 Results

The results obtained with this module are threefold:

- expressiveness
- speedups obtained from the splitting of complex jobs
- power consumption gains when deploying jobs from a mobile device

The last set of results may not be ad relevant to a infrastructure composed of personal computers an d clusters connected by a LAN, but the two first are.

The use of or simple scheduling infrastructure yields the expected speedups (close to optimal) and the user interface allows the definition of complex jobs decomposition, with minimal user burden.

3.5 Heuristic for the allocation of resources in the cloud

The target users of our work have nowadays a new source of computing power: on-demand utility computing infrastructures, e.g. Amazon Elastic Compute Clouds (EC2). With the suitable tasks deployment middleware, these new computing environments, can be used to build clusters capable of executing even the most computing intensive tasks.

If some of the work previously described in this document can solve part of the problems common users have when managing tasks and execution environments, this new computing sources creates a new issue: the need to minimize cost.

In BoT problems, the number of concurrent active tasks needs not be fixed over time because there is no communication among them. To execute tasks on utility computing infrastructures, the number of allocated virtual computers is relevant both for speedups and cost. If too many are created, the speedups are high but this may not be cost-effective; if too few are created, costs are low but speedups fall below expectations. Determining this number is difficult without prior knowledge regarding processing time of each task and job totals.

The relation between the number of allocated hosts, the speedup obtained and the total cost is presented in Figure 3.5, considering CPU time is charged by the hour.

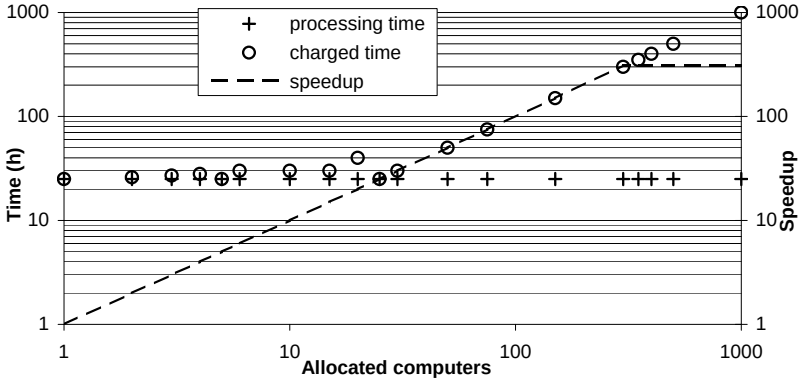

Figure 3.6: Evaluation of cost and speedups

If in order to solve a job comprising 300 tasks (each one with a completion time of about 5 minutes), we allocate 300 computers, then, the maximal speedup is achieved, but at the expense of very high charged time as well. If the minimum time unit charged is one hour, the user will be charged for 300 hours for only 25 hours of total processing time used.

This example clearly shows that when paying the minimum possible (25 hours), the best possible speedup is achieved using 25 computers, i.e., each computer spending one hour solving tasks. When using fewer computers, although speedups are lower, there is no actual decrease in the charged time. When more than 25 computers are allocated, the cost increase is linear w.r.t. speedup improvement.

In addition to the minimum value charged, the maximum possible speedup must also be taken into account. In the example, with 300 similar tasks, the maximum possible speedup is 300, achievable if 300 machines are allocated. Past this value, the addition to the computation of more machines does not yield a higher speedup.

Domestic users or even scientists without Grid or Cluster access would be willing to execute their jobs with substantial speedups while within constrained budgets, instead of paying for the maximum speedup possible. By varying the number of allocated hosts inside the interval one can get higher or lower speedups, depending on the available budget, but guaranteeing that one gets the maximum performance that value could pay.

3.5.1 Contribution

In order to maximize the speedups obtained while guaranteed the minimum possible payment we developed an heuristics that determines how many machines to allocate in utility computing infrastructure.

This heuristic would be trivial if the every task processing time was previously know:

$$n\_machines = \frac{1}{charging\_unit} \sum task\_execution\_time \tag{3.1}$$

Unfortunately each task execution time is only known after it finishes, and time estimates are not precise enough [32]: i) users are not capable of determining each tasks execution time, and ii) small variations on the tasks parameter can lead to high variation on the total execution time.

So, the developed heuristics determines the best number of hosts to allocate in order to guarantee that the payment does not exceed the minimum possible payment, while obtaining the best possible speedup (for the predetermined number of machines).

A simple implementation would calculate the optimal number of machines to allocate, taking into account the average executing time of terminated tasks, as presented inListing **??**.

```
1   remainingTasks --
2   finishedTasks ++
3   totalProcessingTime += concludedTask.processingTime
4   tasksAverageTime = totalProcessingTime/finishedTasks
5
6   possibleTasks = 0
7   for each runningComputer:
8       possibleTasks += runningComputer.possibleTasks(tasksAverageTime)
9
10  necessaryComputers = round(((remainingTasks-possibleTasks) * tasksAverageTime) /
11                             hostProcessingTime)
```

Listing 3.3: Simple heuristic pseudocode executed when a tasks conclude label

Lines 1–4 update and calculate the number of finished tasks, how many tasks are yet to be executed, and the average executing time of those already terminated. On lines 6–8, for each running

machine, the method **possibleTasks(tasksAverageTime)** calculates how many tasks can still be executed until a charging unit is reached.

With the predicted total executing time of the remaining tasks (**(remainingTasks-possibleTasks) * tasksAverageTime)** lines 10–11 how many machine are necessary, besides the ones already running.

Although correct, the direct application of this code, could lead to either one of the following cases: i) allocation of too few hosts if first tasks were of short duration, ii) allocation o too many hosts if the first task was longer than the average.

To solve these problems, we apply two correction factors.

- Tasks are randomly selected for execution, guaranteeing that, even if frists tasks had "abnormal execution duration", their duration would not interfere with the number of allocated hosts
- The number of the calculated number of hosts to allocate is corrected by a correction factor that, as new tasks terminate, increases and converges to 1. This way if the first tasks are longer that the average no more hosts than necessary are allocated.

Periodically the heuristic is executed, so that long running tasks to not delay the calculation of the number of hosts to allocate.

3.5.2 Implementation and results

The heuristic was implemented in Python and evaluated using a discrete event simulator.

The only input the heuristic need is the following:

- number of tasks
- minimum amount of time charged by the infrastructure
- duration af each terminated task

With this information, every time a tasks concludes, a message to the host allocation mechanism is sent.

The heuristic was evaluated with real and synthetic workloads, allowing speedups in line with the number of allocated computers, while being charged approximately the same predefined budget.

3.6 Overall Discussion

There are new user classes requiring computing cycles to solve their problems, furthermore most of their problems fit nicely in the Bag-of-Tasks category. Up until now, most of the work on infrastructures for the execution of parallel tasks did not take into account these users.

In our work we managed to go a step further in order to allow the use of existing cycles by these less favored users.

We worked in three different layers:

- Methods for job definition
- Middleware for deployment on physical infrastructures
- Mechanisms for efficient resource evaluation

With respect to job definition methods, the developed work allows the efficient creation of Bag-of-task for users with little or no knowledge of programming, either with SPADE or Mercury.

SPADE allows the definition of jobs where the execution code is a complete application executable, solely requiring the definition of the application arguments. Although of a simple interface SPADE allow some complex data splitting among tasks.

Mercury requires users to know a programming language, but releases them of the burden of learning tasks creation and data communication APIs. The user writes a sequential application (following a simple template) and Mercury splits the application allowing the execution of the lengthy methods in parallel in different processors. The problem being solved must also fit into Bag-of-Tasks (with no communication between tasks) but our solution allows the development o a single sequential application version that can be later deployed in diverse parallel architectures.

Other lines of work consisted on the adaptation of existing parallel infrastructures to the new user classes without access to high end parallel computing infrastructures.

We developed nuBOINC that by extending BOINC allows any user to create tasks to be executed on the edge of the internet. In order to ease the development of tasks and to increase donor base, the core tasks execution engines are commodity off-the-shelf applications. Users wishing to donate cycles to others just select what applications are allowed be used by others on their computers, thus indirectly stating what kind of work are willing to execute and what problems are helping to solve. The definition of the jobs follows a simple mechanism similar to the one presented in SPADE.

With SPADE the deployment of tasks over a cluster of computers has also become accessible to users with low computer proficiency, just requiring the installation of a server application and simple configuration steps.

Although the use of Utility Computing has been proposed to the execution of parallel jobs, that was just used in the execution of pure MPI applications with a fixed number tasks. With Bag-of-Tasks problems the number of concurrent tasks is not fixed and influences the obtained speedup. In a utility computing infrastructure, this variation of the number of tasks also leads to different final payment. In order to maximize the speedup for a fixed payment amount we developed a heuristic to determine

the optimal number of hosts to allocate. This value is calculated without any user intervention and allows a close to optimal use of the payment and allocated machines.

Besides the development of this heuristics, an easier integration of new and diverse resources was also accomplished.

A new algebra for the definition of requirements was developed along with an extensible middleware for their evaluation. Users are now allowed to express their computing requirements with the guarantees that the results are precise (with respect to the real available resources) and that a useful sorting of available sources is produced. Besides the current evaluated resources, the developed middleware allows the injection of new evaluation code, thus allowing the inclusion of new resources to be evaluated (either by an ordinary user or by an administrator) without any system operation interruption. The use of different algebras for different user classes also allows a better fitting of the available resources to the real needs: some users have their requirements evaluated using bool logic while other have their requirement with more permissive fuzzy logic operators.

4 Conclusion

In this document we presented the core of the thesis proposal and all the work developed in the course of the PhD being carried out.

In the first Chapter we presented the assumptions that lead this works and the need for the developed work. In previous years new user classes have emerged in the parallel computing arena, with specific needs but for whom existing systems are not fit.

The possible existence of Distributed Computing Systems capable of filling these users needs lead to the study of those systems by means of a new taxonomy (presented in Chapter 2). This study lead to the conclusion that such system does not exist and allowed the extraction of some fundamental characteristics.

The previous chapter describes the works carried out in order to try to bring parallel computing and the necessary computing cycles to those new user classes. We presented the overall working areas (job submission, resource integration an discover, and Jobs deployment) and detailed the developed work stating what problems each developed modules solves and how.

The main contributions of the work carried out are as follows:

- Definition of a taxonomy for the characterization and evaluation of Internet based Distributed Computing Systems [65]
- Development of heuristic for the allocation of machines in a Utility computing infrastructure [63, 59]
- Definition of an algebra for the precise resource and services discovery and evaluation [57]
- Design and implementation of a middleware for automatic and transparent parallelization of Bag-of-Tasks [62]
- Design and implementation of an user interface for the easy definition of parallel jobs [61]
- Design and implementation of a middleware for the distribution of tasks over a personal cluster [61]
- Design and implementation of a middleware for the distribution of tasks over the Internet with Comunity based Cycle sharing [60]

Further details on each contribution can also be found in the publications where they were presented (already made available).

Bibliography

[1] D. Abramson, R. Sosic, J. Giddy, and B. Hall. Nimrod: A tool for performing parametised simulations using distributed workstations. In *The 4th IEEE Symposium on High Performance Distributed Computing*, August 1995.

[2] Amazon Web Services LLC. Amazon elastic compute cloud (amazon ec2), 2009. http://amazon.com/ec2.

[3] Y. Amir, B. Awerbuch, and R. S. Borgstrom. A cost-benefit framework for online management of a metacomputing system. In *ICE '98: Proceedings of the first international conference on Information and computation economies*, pages 140–147, New York, NY, USA, October 1998. ACM.

[4] Y. Amir, B. Awerbuch, and R. S. Borgstrom. The java market: Transforming the internet into a metacomputer. Technical report, Johns Hopkins University, 1998.

[5] D. P. Anderson. Local scheduling for volunteer computing. *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8, 26-30 March 2007.

[6] D. P. Anderson and G. Fedak. The computational and storage potential of volunteer computing. In *IEEE/ACM International Symposium on Cluster Computing and the Grid*, May 2006.

[7] D. P. Anderson, E. Korpela, and R. Walton. High-performance task distribution for volunteer computing. In *E-SCIENCE '05: Proceedings of the First International Conference on e-Science and Grid Computing*, pages 196–203. IEEE Computer Society, 2005.

[8] H. E. Bal, A. Plaat, T. Kielmann, J. Maassen, R. van Nieuwpoort, and R. Veldema. Parallel computing on wide-area clusters: the albatross project. In *In Extreme Linux Workshop*, pages 20–24, 1999.

[9] J. E. Baldeschwieler, R. D. Blumofe, and E. A. Brewer. Atlas: an infrastructure for global computing. In *EW 7: Proceedings of the 7th workshop on ACM SIGOPS European workshop*, pages 165–172, New York, NY, USA, 1996. ACM.

[10] A. Baratloo, M. Karaul, H. Karl, and Z. M. Kedem. Knittingfactory: An infrastructure for distributedweb applications. Technical Report TR 1997ñ748, Courant Institute of Mathematical Sciences - New York University, November 1997.

[11] A. Baratloo, M. Karaul, H. Karl, and Z. M. Kedem. An infrastructure for network computing with Java applets. *Concurrency: Practice and Experience*, 10(11–13):1029–1041, 1998.

[12] A. Baratloo, M. Karaul, Z. M. Kedem, and P. Wijckoff. Charlotte: metacomputing on the web. *Future Gener. Comput. Syst.*, 15(5-6):559–570, 1999.

[13] J. Berntsson. G2dga: an adaptive framework for internet-based distributed genetic algorithms. In *GECCO '05: Proceedings of the 2005 workshops on Genetic and evolutionary computation*, pages 346–349. ACM, 2005.

[14] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, 1995.

[15] T. Brecht, H. Sandhu, M. Shan, and J. Talbot. Paraweb: towards world-wide supercomputing. In *EW 7: Proceedings of the 7th workshop on ACM SIGOPS European workshop*, pages 181–188. ACM, 1996.

[16] P. Cappello, B. Christiansen, M. Neary, and K. Schauser. Market-based massively parallel internet computing. In *Third Working Conf. on Massively Parallel Programming Models*, pages 118–129, Nov 1997.

[17] P. Cappello and D. Mourloukos. Cx: A scalable, robust network for parallel computing. *Scientific Programming*, 10(2):159–171, 2002.

[18] J. Celaya and U. Arronategui. Ya: Fast and scalable discovery of idle cpus in a p2p network. In *GRID '06: Proceedings of the 7th IEEE/ACM International Conference on Grid Computing*, pages 49–55. IEEE Computer Society, September 2006.

[19] B. O. Christiansen, P. Cappello, M. F. Ionescu, M. O. Neary, K. E. Schauser, and D. Wu. Javelin: Internet-based parallel computing using java. *Concurrency: Practice and Experience*, 9:1139–1160, 1997.

[20] M. Dharsee and C. Hogue. Mobidick: a tool for distributed computing on the internet. *Heterogeneous Computing Workshop, 2000. (HCW 2000) Proceedings. 9th*, pages 323–335, 2000.

[21] U. Egede, K.Harrison, R. Jones, A. Maier, J. Moscicki, G. Patrick, A. Soroko, and C. Tan. Ganga user interface for job definition and management. In *Proc. Fourth International Workshop on Frontier Science: New Frontiers in Subnuclear Physics*, Italy, September 2005. Laboratori Nazionali di Frascati.

[22] S. Esteves, L. Veiga, and P. Ferreira. Gridp2p: Resource usage in grids and peer-to-peer systems. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, April 2010.

[23] G. Fedak, C. Germain, V. Neri, and F. Cappello. Xtremweb: a generic global computing system2001. *Cluster Computing and the Grid, 2001. Proceedings. First IEEE/ACM International Symposium on*, pages 582–587, 2001.

[24] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, 1997.

[25] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. S. Sunderam. Pvm: Parallel virtual machine. a users' guide and tutorial for networked parallel computing. Cambridge, MA, USA, 1994. MIT Press.

[26] C. Germain, V. Néri, G. Fedak, and F. Cappello. Xtremweb: Building an experimental platform for global computing. In *GRID '00: Proceedings of the First IEEE/ACM International Workshop on Grid Computing*, pages 91–101. Springer-Verlag, 2000.

[27] P. A. Gray and V. S. Sunderam. Icet: Distributed computing and java. *Concurrency - Practice and Experience*, 9(11):1161–1167, 1997.

[28] R. Gupta and V. Sekhri. Compup2p: An architecture for internet computing using peer-to-peer networks. *IEEE Trans. Parallel Distrib. Syst.*, 17(11):1306–1320, 2006.

[29] M. Karaul. *Metacomputing and Resource Allocation on the World Wide Web*. PhD thesis, New York University, May 1998.

[30] T. Kielmann, H. E. Bal, J. Maassen, R. van Nieuwpoort, R. Veldema, R. Hofman, C. Jacobs, and K. Verstoep. The albatross project: Parallel application support for computational grids. In *In Proceedingof the 1st European GRID Forum Workshop*, pages 341–348, 2000.

[31] G. Kouretis and F. Georgatos. Livewn, cpu scavenging in the grid era. http://arxiv.org/abs/cs/0608045, arXiv.org,, August 2006.

[32] C. B. Lee, Y. Schwartzman, J. Hardy, and A. Snavely. Are user runtime estimates inherently inaccurate? In *Job Scheduling Strategies for Parallel Processing, 10th International Workshop, JSSPP 2004*. Springer, 2005.

[33] U. Leiden. Leiden classical. http://boinc.gorlaeus.net/.

[34] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7:321–359, November 1989.

[35] M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th Intl.Conf. of Distributed Computing Systems*. IEEE Computer Society, June 1988.

[36] A. Luther, R. Buyya, R. Ranjan, and S. Venugopal. Alchemi: A .net-based enterprise grid computing system. In *Proceedings of the 6th International Conference on Internet Computing (ICOMP'05)*. CSREA Press, Las Vegas, USA, June 2005.

[37] A. Luther, R. Buyya, R. Ranjan, and S. Venugopal. *High Performance Computing: Paradigm and Infrastructure*, chapter Peer-to-Peer Grid Computing and a .NET-based Alchemi Framework. Wiley Press, USA, June 2005.

[38] R. Mason and W. Kelly. Peer-to-peer cycle sharing via .net remoting. In *AusWeb 2003. The Ninth Australian World Wide Web Conference*, 2003.

[39] R. Mason and W. Kelly. G2-p2p: a fully decentralised fault-tolerant cycle-stealing framework. In *ACSW Frontiers '05: Proceedings of the 2005 Australasian workshop on Grid computing and e-research*. Australian Computer Society, Inc., 2005.

[40] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical report, University of Tennessee, Knoxville, TN, USA, 1994.

[41] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon. Top 500 supercomputers sites, November 2010. http://www.top500.org/.

[42] H. Mikkonen. Enabling computational grids using jxta-technology. In *Peer-to-peer technologies, networks and systems - Seminar on Internetworking*. Helsinki University of Technology, 2005.

[43] M. O. Neary, S. P. Brydon, P. Kmiec, S. Rollins, and P. Cappello. Javelin++: scalability issues in global computing. In *JAVA '99: Proceedings of the ACM 1999 conference on Java Grande*, pages 171–180. ACM, 1999.

[44] M. O. Neary, A. Phipps, S. Richman, and P. R. Cappello. Javelin 2.0: Java-based parallel computing on the internet. In *Euro-Par 2000, Parallel Processing, 6th International Euro-Par Conference, Munich, Germany, August 29 - September 1, 2000, Proceedings*, 2000.

[45] N. Nisan, S. London, O. Regev, and N. Camiel. Globally distributed computation over the internet - the popcorn project. In *ICDCS '98: Proceedings of the The 18th International Conference on Distributed Computing Systems*, page 592. IEEE Computer Society, 1998.

[46] L. Oliveira, L. Lopes, and F. Silva. P^3: Parallel peer to peer - an internet parallel programming environment. *Lecture Notes in Computer Science*, 2376, 2002.

[47] K. Pearson. Distributed computing. http://www.distributedcomputing.info/.

[48] H. Pedroso, L. M. Silva, and J. G. Silva. Web-based metacomputing with JET. *Concurrency: Practice and Experience*, 9(11):1169–1173, 1997.

[49] P. Rodrigues, C. Ribeiro, and L. Veiga. Incentive mechanisms in peer-to-peer networks. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–8, April 2010.

[50] L. F. G. Sarmenta. *Volunteer Computing*. PhD thesis, Massachusetts Institute of Technology, June 2001.

[51] L. F. G. Sarmenta and S. Hirano. Bayanihan: building and studying Web-based volunteer computing systems using Java. *Future Generation Computer Systems*, 15(5–6):675–686, 1999.

[52] K. E. Schauser, C. J. Scheiman, G.-L. Park, B. Shirazi, and J. Marquis. Superweb: Towards a global web-based parallel computing infrastructure. In *IPPS '97: Proceedings of the 11th International Symposium on Parallel Processing*, pages 100–106. IEEE Computer Society, 1997.

[53] B. Schmidt. A survey of desktop grid applications for e-science. *International Journal of Web and Grid Services*, 3(3):354–368, 2007.

[54] V. Sekhri. Compup2p: A light-weight architecture for internet computing. Master's thesis, Iowa State University, Ames, Iowa, 2005.

[55] K. Shudo, Y. Tanaka, and S. Sekiguchi. P3: P2p-based middleware enabling transfer and aggregation of computational resources. *Cluster Computing and the Grid, 2005. CCGrid 2005. IEEE International Symposium on*, 1:259–266, May 2005.

[56] J. N. Silva. Optimização e avaliação de aplicações de simulações em sistemas paralelos. Master's thesis, Instituto Superior Técnico, 2003.

[57] J. N. Silva, P. Ferreira, and L. Veiga. Service and resource discovery in cycle-sharing environments with a utility algebra. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1 –11, 2010.

[58] J. N. Silva and P. Guedes. Ship hull hydrodynamic analysis using distributed shared memory. In *Applied Parallel Computing, New Paradigms for HPC in Industry and Academia, 5th International Workshop, PARA 2000 Bergen, Norway, June 18-20, 2000 Proceedings*, volume 1947 of *Lecture Notes in Computer Science*, pages 366–372. Springer, 2000.

[59] J. N. Silva, L. Veiga, and P. Ferreira. Heuristic for resources allocation on utility computing infrastructures. In *Proceedings of the 6th international workshop on Middleware for grid computing*, MGC '08, pages 9:1–9:6, New York, NY, USA, 2008. ACM.

[60] J. N. Silva, L. Veiga, and P. Ferreira. nuboinc: Boinc extensions for community cycle sharing. *Self-Adaptive and Self-Organizing Systems Workshops, IEEE International Conference on*, 0:248–253, 2008.

[61] J. N. Silva, L. Veiga, and P. Ferreira. Spade: scheduler for parallel and distributed execution from mobile devices. In *Proceedings of the 6th international workshop on Middleware for pervasive and ad-hoc computing*, MPAC '08, pages 25–30, New York, NY, USA, 2008. ACM.

[62] J. N. Silva, L. Veiga, and P. Ferreira. Mercury: a reflective middleware for automatic parallelization of bags-of-tasks. In *Proceedings of the 8th International Workshop on Adaptive and Reflective MIddleware*, ARM '09, pages 1:1–1:6, New York, NY, USA, 2009. ACM.

[63] J. N. Silva, L. Veiga, and P. Ferreira. A2ha - automatic and adaptive host allocation in utility computing for bag-of-tasks. Submited for publication on JISA - Journal of Internet Services and Applications, Springer, 2010.

[64] L. M. Silva, H. Pedroso, and J. G. Silva. The design of jet: A java library for embarrassingly parallel applications. In A. Bakkers, editor, *Parallel programming and Java: WoTUG-20*, pages 210–228. IOS Press, 1997.

[65] J. N. Siva. Volunteer computing and public cycle sharing systems. Report for the Curricular Unit *Reserach Topics*, October 2010.

[66] M. Somers. Leiden grid infrastructure, 2007. http://fwnc7003.leidenuniv.nl/LGI/docs/.

[67] L. Veiga, J. N. Silva, and J. C. Garcia. *Peer4Peer: E-science Communities for Overlay Network and Grid Computing Research*. Springer, 2011.

[68] L. V. Veiga, R. Rodrigues, and P. Ferreira. Gigi: An ocean of gridlets on a "grid-for-the-masses". In *7th IEEE International Symposium on Cluster Computing and the Grid (CCGrid07)*, May 2007.

[69] J. Verbeke, N. Nadgir, G. Ruetsch, and I. Sharapov. Framework for peer-to-peer distributed computing in a heterogeneous, decentralized environment. In *GRID '02: Proceedings of the Third International Workshop on Grid Computing*, pages 1–12. Springer-Verlag, 2002.

[70] D. Zhou and V. Lo. Cluster computing on the fly: Resource discovery in a cycle sharing peer-to-peer system. In *IEEE International Symposium on Cluster Computing and the Grid*, 2004.