Volunteer computing and public cycle sharing systems.

João Nuno Silva

October 27, 2010

Abstract

In recent years there has been a significant development and research on Distributed Computing Systems to allow the public execution of embarrassingly parallel jobs on the Internet. The work developed is very broad and ranges from the definition of adequate programming techniques, to new network architectures, or even to more efficient scheduling techniques.

Although currently a myriad of systems exist, employing different technologies, few are able to attract a suitable user base. The only Distributed computing platform with widespread use is BOINC and its derivatives. All other systems are in effect not widely used.

In this paper we present a new taxonomy for the characteristics of Distributed Computing Systems. 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. We conclude this document 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.

1 Introduction

The Internet is a good source of computational power for the execution of parallel tasks: all of the connected computer are idle some of the time and are perfectly capably of executing most available jobs.

Taking this into account, during recent years, there has been work on the development of tools and systems to leverage these available resources. The goals of these systems are two fold: i) allow users with parallel jobs to deploy them to be executed on remote computers, and ii) attract owners of connected computers to donate processing time to those jobs that need it.

These goals are to be obtained with minimal burden to those that intervene in the process: programmers and donors. Programmers should have

minimal work parallelizing the applications to be executed on the Internet and should gain from the parallel execution of their tasks. Issues like reliability of the returned values and security of data and code should also be handled by the system.

On the other hand the donor should be disturbed to a minimum, when installing the system, and executing the parallel code: i) the installation should be straightforward, ii) the security should not be compromised, and iii) the overhead incurred from the download and execution of the code should be minimum.

The way each system handles and solves the previous problems is fundamental to its widespread adoption as a valid solution to the execution of lengthy Bag-of-Tasks (BoT) problems.

The problems that fit to the Bab-of-Tasks class are composed of various independent tasks. These are usually embarrassingly parallel problems, whose tasks share the same code, but have different input data or execution arguments. This class of problems are well fit to the internet environment, as no coordination between donors is needed and because efficient mechanisms can be applied to tackle donors volatility, and potential security faults. On the Internet, these Distributed Computing Systems, implement both the Volunteer Computing or Cycle Sharing paradigm, where users on the edge of the internet donated idle CPU time for the execution of remote code.

1.1 Document Roadmap

In the following subsection we present some previous work that tried to systematize the characteristics of distributed and parallel computing systems.

Section 2 presents the taxonomy for the characteristics of cycle-sharing systems and how each analyzed system fits into it. In Section 3 we present the most relevant characteristics that affect user adhesion, present the best solutions and try to conclude how those affect the installed user bases.

The document closes with the conclusions.

1.2 Related Work

Although there is some work trying to systematize the characteristics of distributed computing infrastructures [1, 2, 3], the main focus has been on network and software architectural decisions, not including the various security aspects and user level interaction models. Furthermore, these surveys also describe systems not usable in a public infrastructure (with admission of donors or clients), while this document will focus on Internet based free access systems. The range of systems presented in our document is much broader (presenting more systems, and from a wider temporal range) than those presented in existing documents.

Other documents also describe some of the characteristics dealt with in our survey. Marcelo Lobosco et al. [4] present a series of systems and libraries that allow the use of Java in High Performance Computing scenarios. The document is mostly focused on the description of programming paradigms available for the development of parallel applications in Java. The described systems are, however, mostly targeted to cluster environments.

Koen Vanthournout et al. [5] describe currently available resource discovery mechanisms. This work describes the different available peer-to-peer network topologies and service discovery frameworks. While the presented mechanisms can be targeted to distributed computing systems, the presented application examples are mostly included in the file sharing category.

2 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 client 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 [6]. 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 [7, 8]
- **1996** : ParaWeb [9]
- **1997** : IceT [10], SuperWeb [11] Charlotte and KnittingFactory [12, 13, 14, 15], JET [16, 17] and Javelin [18, 19, 20, 21]
- **1998** : Java Market [22, 23] and POPCORN[24]
- **1999** : Bayanihan [25, 26]

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 [27] and XtremWeb[28, 29]
- **2002** : JXTA-JNGI [30, 31], P^3 [32] and CX [33]
- **2003** : G2-P2P [34, 35]
- **2004** : CCOF [36]
- **2005** : Personal Power Plant [37], CompuP2P [38, 39], G2DGA [40], Alchemi [41, 42] and BOINC [43, 44, 45]
- **2006** : YA [46]
- **2007** : Leiden[47, 48], Ginger/NuBoinc[49, 50, 51, 52]

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 [53, 54] 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 [55] 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 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 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

Point-to-point: In a point-to-point topology: the user having work to be executed must own and operate his own server. The creation of jobs must be performed in that server. At a later instant, computers owned by the donors contact directly the servers owned by the clients (those needing processing time and creating work to be done).

As it is the responsibility of the user submitting the jobs to install and maintain all the hardware and software necessary to provide work to the donors, this simple solution adds burden to the client user. Furthermore there is no reuse of previously installed infrastructures.

On the donor side, this solution is not very flexible. This user has to exactly know the identification of the server where certain jobs are hosted, and it is impossible to load balance the requests to distinct computers.

Single Server: By decoupling the place where jobs are created (client computer) and the server where they are stored, it becomes possible for a single server to hosts projects and jobs from users geographically dispersed. This solution allows the reuse of the infrastructure, and the reuse of the processing code. Users can now submit different data to be processed by previously developed code.

This architecture requires the existence of remote job creation mechanisms. The existence of a simple user interface is enough.

This architectural solution still requires, as in a point-to-point architecture, the knowledge of the exact identification of the server hosting the jobs to be executed.

Directory Server: With the addition of a directory server, the problems with the point-to-point and single server approaches, with respect to server identification and location, are solved. The donors contact a directory server that redirects requests to one of the servers supplying jobs. Although the donor still has to know the identification of a computer (the directory server), this server will act as an access point to various work sources.

The architecture of the underlying server may follow the patterns presented earlier (single server or point-to-point). Either the servers are shared among different clients by allowing the creation of jobs from a remote location, or a server can only be used by its owner. Furthermore all communication is still performed directly between the donor computer and the server where jobs are stored.

Even with a directory server, scalability and availability problems continue to exist. For each job, there is still only one server hosting its tasks. All donors wanting to execute such tasks contact the same server, that may not have enough resources to serve all requests, in particular to transfer all data efficiently. Also, in case of failure there are no recovery options, being impossible for any donor to execute tasks from jobs hosted on such failed server.

Hierarchical Servers: In order to tackle scalability issues, the simplest way is to split data among several servers. In the case of Distributed Computing systems, this division can be made at different granularity levels: at the job level (different jobs hosted on different servers) or at the task level (different tasks on different hosts).

In order to manage a set of systems, the simplest solution is to organize them in a tree structure. This hierarchical structure can range from the simple two tier architecture (with one coordinator and a set of servers), to a deeper organization.

The most evident use of a hierarchical infrastructure is to load balance server load. Taking this into account, when jobs (or tasks) are created, the server responsible for them can be the one with less load. In this case, a job (and all its tasks) can be hosted on a server or have its tasks distributed among several servers.

Systems that allow the creation of recursive tasks (allowing tasks themselves to spawn new tasks) also fit well in a hierarchical architecture. When tasks are created, a server to host the tasks is selected. In order to maintainn information about dependencies, it is necessary that each server maintains a list of the servers hosting the sub-tasks. This requires a tree-based architecture, based on the task dependencies.

As each one of the servers stores parts of the jobs, some level of load balancing is possible and scalability attained. Nonetheless even with this solution, availability is still an issue, as if a server crashes the work stored

there becomes unavailable.

Replicated Servers The replication of task data among several servers addresses both availability and scalability issues.

In systems with replicated servers the information about a single task is stored in more than one server. When submitting work, the client contacts one server, being transparent to him how the data is stored. Later, when donors contact the system in search of work, one copy of a task is retrieved.

The replication can be performed on two different levels: server and task. In the first case, all the data stored in a server is replicated on other servers. If the replication is at the task level, there is no strict mapping between the information stored on the server, replicas of tasks from the same jobs can be stored on different servers.

In the case of failure of a server, as replicas of that data are stored elsewhere, the system continues operational, maintaining all task information accessible. Furthermore, with this solution donors do not have to contact the same set of servers to retrieve tasks from a given job, which also tends to balance load across servers.

Besides the placement of the replicated tasks data, the management of the execution of those tasks is also a complex issue. If tasks are idempotent, the multiple execution of the same task causes no harm; thus if the system only allows such type of tasks the replication implementation is straightforward.

If tasks are not idempotent, it is necessary to guarantee that before starting a task, no replica of it was previously started. These verification mechanisms and the communication overhead may hinder the gains from replication, in the case of shorter tasks.

There is no strict relation between replicated storage and replicated execution of tasks. In section 2.2 we present how replicated execution of tasks is performed and its uses.

Peer-to-peer The most distinct characteristic of a peer-to-peer architecture is that any intervening computer (peer) performs at least two simultaneously, yet distinct roles: i) a peer is a donor, by executing tasks submitted by others, and ii) acts as a server because it stores data, results, and helps on work distribution. Furthermore a intervening computer can also act as a client, from where jobs and tasks are created.

While on other systems, dedicated servers are used to store data, in a peer-to-peer system, the data is on the edge of the Internet on often insecure, unreliable computers. This problem requires more resilient solutions than on other systems: i) the distribution of the tasks data in different nodes is essential not to overload a single computer, and ii) replication of data should be implemented to guarantee that the (highly probable) failure of a

peer does not compromise the execution of a job.

With respect to donors' high volatility, the problems on a peer-to-peer infrastructure are similar to those observed in systems relying on dedicated servers. These issues will be described in Section 2.2.

The entry point for a peer to peer network can be any one of the already participating nodes, furthermore any peer can only know (and contact with) a limited number of peers. These issues affect normal operation of the system: the way discovery and selection of tasks is performed, how remote resources are discovered, and how information is exchanged between the peer acting as client and the one acting as donor. These issues will be detailed in the next sub-sections.

2.1.2 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

Centralized Database The use of a centralized database is the simplest of the resource evaluation methods. When a donor registers for the first time on the system, information about the available resources is stored in a database.

This solution has a few important drawbacks: only a single server is allowed, and the rate with which resource availability changes may not be too high.

Even if several servers are used to store tasks information, the use of a centralized database overrides any gain from the use of multiple servers (as described previously). There is one central failure point, reducing both availability and scalability.

If the resources available on the donor change with a high frequency, the rate at which the database has to be updated may not scale to large number of donors.

If none of the previous issues are problematic the use of a single database is effective and offers efficient resource discovery, and subsequently more efficient job execution and resource usage.

Decentralized Database The use of a decentralized database for storing the available resources information, not only solves some of above problems but also fits nicely on peer-to-peer systems, or those using multiple servers.

In the same way as data can be split among several computers, to increase availability and scalability, so the distribution of the information about donors can be optimized: the burden of selecting a donor is distributed, and the failure of a server doe not become catastrophic.

On systems with a single database, update frequency can become a problem due to the network traffic it may generate. With a distributed database, although information still needs to be transferred, it can be done to a server closer to the donor: the bottleneck of having a single server is avoided, and the messages have to transverse few network links.

The information stored in the distributed database, can be either replicated or partitioned. If the information is replicated, some inconsistency among replicas can be tolerated: in this case the only drawback is the possible selection of non optimal hosts to execute a task.

Polling If it is impossible to maintain a database (either centralized or decentralized) the only solution to discover the resources characteristics at a given moment, is by directly polling donors to gather information about their available resources.

With an efficient donor discovery infrastructure, polling those computers is a straightforward step, but requiring extra care to guarantee close to optimal answers without too much network bandwidth consumption. To select the optimal donor for a task it would be necessary to poll every donor available, solution that is impractical.

While with a database, it is possible to have a overall vision of the available resources, allowing the selection of the best donors; with polling that becomes difficult. A complete vision of available resources requires a full depth search, impossible for large systems. Thus only a partial view of the system is possible at a given moment, and the selection of the donors may not yield the optimal answer.

So, the use of polling to evaluate remote resources guarantees that the information about contacted hosts is up-to-date, but does not guarantee that the best available host is used to execute every tasks.

2.1.3 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

Eager When an eager scheduling policy is used, the selection of hosts to execute available tasks is blind, neither taking into account the characteristics of the computer nor the possible task execution requirements.

Whenever a donor host is idle and requests some work, the system assigns it a task: randomly selecting it or using a FIFO policy.

Some level of fairness can be guaranteed (by first assigning previously created tasks) but offers no guarantee about the optimal resource usage. This solution is the simpler to implement as no global information about available resources is needed.

Resource Aware In order to optimize the execution of tasks, a resource aware scheduling policy assigns tasks to the more capable machines before assigning tasks to other less capable.

This solution still picks tasks to be executed either randomly or in a FIFO manner, but then, for the task in question, it selects a machine with enough resources (e.g. memory or CPU speed). This selection is made taking into account task requirement information.

To implement this solution it is required the existence of a database (where servers query for the best donor), or polling donors to discover their resources.

This donor selection method guarantees that every task requirements are met and that those tasks are executed close to minimum possible execution time. On the other hand, this method does not minimize the overall makespan of a set of tasks or job, as different donor assignments (with delay on a task starting) and task execution order could lead to a best overall performance.

Heuristic When selecting execution hosts, previous policies select tasks by a predefined order (or randomly) and just take into account each task individual resource requirements.

With a complete knowledge of tasks requirements and available resources it would be possible to schedule tasks to the best hosts in order to minimize every job makespan. In a highly dynamic environment, this task is impossible: i) available resources are not known when submitting tasks, and ii) the exact execution times for each tasks is often impossible to predict.

To reduce a job makespan it is necessary to use heuristics. These empirical rules, although not providing the optimal solution, allow better jobs execution times than a blind or resource aware scheduling.

Market Oriented Previous scheduling policies resort to the information about available computational resources for assigning tasks to donors. This limits any user intervention on the selection of the hosts to execute tasks.

This host and task matching can be changed and manipulated by both clients and donors if a resource market exists.

In this new market driven environment, the selection of the tasks to be executed is made taking into account some bidding mechanism and using some sort of currency earned by executing other peoples' jobs.

Clients state how much they are willing to pay for the execution of their tasks, donors decide the cost of their resources, and a matching algorithm (implementing available bidding mechanisms) matches the tasks with the donors.

Buya et al [56] present an overview of market oriented mechanisms presented on current grid systems. In this document the currently used auction mechanisms are presented: i) english auction, ii) dutch auction, and iii) Double Auction.

While the english auction method is the mostly used mechanism (where buyers increase the value they are willing to pay), its straight implementation in a wide area distributed system has a high communication overhead. The Vickrey [57] method is an efficient auction methodology that can replace the english action, thus reducing the communication overhead. In a Vickrey auction, buyers bid the product by stating the highest value they are willing to pay, the winner of the auction is the one that bid with the highest value, but only pays the value of the second highest bid plus one monetary unit.

User Priority Although the selection of the hosts to run a task is important, another fundamental issue to scheduling efficiency is task selection. Besides matching tasks to suitable hosts, on the server side it is necessary to select from which user tasks will be executed from.

Several users can have tasks that compete from the same resources and the system should have means to prioritize those tasks. Heuristic policies can handle the task to execute next, but this selection can be dependent only on the user that submitted the work, assigning each user a different priority. This priority mechanisms can be static, each user has a predefined priority, or they can vary with time. Furthermore, users can be grouped in classes according to resources donated, reputation, etc.

2.1.4 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

Pull vs Push If it is the donor that initiates the request for a new task the system uses a pull mechanism. After completion of a task, the donor contacts one server in order to be assigned more work.

In the case of push, it is the initiative of the server storing information about a task to initiate it. A donor is selected, contacted, and the task data is transferred.

The pull mechanism is more efficient when the donor also performs some sort of local scheduling: by selecting the servers to contact and projects to execute. Depending on the execution time spent on the various jobs, it is the donor that selects where the new tasks come from. Furthermore there is no need for a database to store resources information: the donor, when requesting for work, may inform the server about its available resources.

If there is a centralized database with the resources available on the various donors, the push mechanism can be easily implemented. The server hosting the resources information database knows the characteristics of the donors and with that information can easily assign them tasks.

Direct vs Brokered If the system only has one server the distribution mechanism is necessarily direct, independently of the direction (pull or push). When starting a task, the donor contacts (or is contacted by) a server that can provide task data without the intervention of any other server.

In the case of a peer-to-peer architecture or when using several servers, the donor may not contact directly the machine storing the task data.

In the case of a brokered pull, if the job or project is not owned by the server (or peer) contacted or has no tasks capable of being executed, it is responsibility of that server (or peer) to discover a suitable task. The server (or peer) forwards the request and after receiving suitable data delivers it to the donor.

The brokered push mechanism is mostly used in peer-to-peer architectures. The peer storing a task contacts neighbor peers trying to find a

suitable host to execute it. If these contacted peers are incapable (because they are unsuitable or are not idle) they forward the request to another peers.

2.1.5 Analysis

Table 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
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 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 2.3.10, 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 2.3.7), 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.

2.2 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.

2.2.1 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

Code / Data The developer of the code to be executed can have concerns about privacy guarantees of both the code and the data. The algorithms used can be proprietary as well as the data.

The mechanisms to guarantee code privacy are similar to those of data.

By encrypting the communication between the server and the donor it is possible to assure that from an external computer it is not possible to access the downloaded information.

During the execution of the tasks, on the donor computer, the privacy of the data and code should also be preserved. Executing the task inside a sandbox and not storing any information in files outside it can guaranteed that no external malicious process running in the donor can access the code. This solution does not fully prevent the access to the information, as as compromised/modified execution environment (sandbox or virtual machine) has access to whole task state.

Anonymity Another information that can be hidden from the donor is the identity of the client, guaranteeing some sort of anonymity. The information is twofold: i) the real identity of the owner of the jobs, and ii) the purpose of the work.

The implementation of anonymity can be performed on the server side, by not disclosing more information than necessary.

2.2.2 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

Executable Verification The simplest method to verify if the results were produced by a non tampered application is to perform some sort of executable verification.

The middleware installed at the donor computer calculates the checksum of the executable and compares it with checksum of the correct program (a value previously calculated and stored in the serveror obtained from a trustable source).

This method guarantees that the executed code is the one provided by the client, but still allows the tampering and modification of the result transmitted by the donor.

Spot-checking With spot-checking what is verified is not the correctness of the results, or the executable producing them, but the reliability of the donor.

Systems that use Spot-checking generate quizzes enclosed in dummy tasks, whose results are previously known. These tasks are periodically sent to donor computers, that treat them as regular tasks. Comparing the returned result with the expected one, it is possible to know if results returned by the same host are to be trusted.

This system, still does not guarantee that the results transmitted from the donors are correct: only that the dummy tasks where executed correctly, and that, with a high probably, the donors are reliable, provided that they can not identify dummy tasks among all tasks.

In case of discovery of a donor with incorrect behaviour, the server can take the appropriate measures about the results previously returned by that donor.

Redundancy With redundancy, several instances of the same task are executed on different hosts and the results returned by each execution are compared and used to decide which result is considered correct. Usually, a simple voting is performed and the result obtained by the majority of the executions is the one considered correct (quorum). If a decision is not reached, more instances of that task are launched.

Redundancy can only be used if tasks are idempotent; only in this case it is safe to execute several times the same task. Furthermore, special care should be taken if the system is market driven, as the execution of multiple instances of the same task has a higher cost to the user.

If a compromised donor executes several replicas of the same task, the result can be forged. Thus, the execution of replicas of the same task must be performed on distinct donors.

Reputation In addition to the use of any of the previous techniques, it is possible to implement a reputation scheme, where information about incorrect results is used on the calculation of a donor's reputation.

This reputation value can then be used on the scheduling of tasks, assigning tasks to more reliable donors, or after receiving the result to perform additional result verification.

Although a reputation scheme may not fully guarantee the result integrity, it may help increase system performance by allowing the execution of tasks on more reliable and, non compromised, donors.

2.2.3 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

Redundancy Besides in result integrity checking, redundancy can also be used in order to increase reliability of a system.

By launching several instances of the same task, the probability that at least one tasks concludes increases, in the case of donor failure. With redundancy, the number of tasks to launch is fixed. Even if the first task concludes in the expected timeframe, redundant tasks (these cases, unnecessary) were still created and partially executed. This drawback is overridden if redundancy is also used to guarantee result integrity.

When using redundancy, slower computers may not be rewarded for the execution of tasks, as before these slower computers returned the results, a faster machine had already done that.

Restarting To solve the blind execution of replicas of the same task, a small modification to the replica launch decision mechanism can be made.

Only after all tasks have been launched at least once, and when some results are still due, the system decides to relaunch tasks: those not finished can be restarted.

Again, as in redundancy, either the previously started instance or the new one will eventually finish yielding the expected result: either the first task was executed on a slow computer, or that donor went off line or failed.

Checkpointing Both redundancy and restarting have problems when tasks are not idempotent or there is some form of payment for the execution of tasks, or in the case of large-running tasks: i) non idempotent tasks can not be executed multiple times, and ii) users may not be willing to pay for a task more that the real execution time.

To solve these issues it is necessary to guarantee that no piece of a task is executed twice. So in case a donor crashes, the tasks need to be restarted from the last correctly executed instruction. To do so, some sort of checkpointing should be periodically performed.

Besides the idempotence and payment concerns, by using checkpointing, the conclusion of restarted tasks is faster, as no duplicate instructions are executed, at the expense of increased time spent to take and store periodic checkpointing of the tasks.

2.2.4 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

User Trust The more lax mechanism is the simple trust on the users submitting the tasks. No external verification mechanisms, as donors believe in the integrity of the programmers that developed the downloaded code, and that no harm comes from executing it.

This requires a that both the owner of the server, task creator, and purpose of the code to be well know by the potential donors.

Executable inspection In order to execute tasks in a non modified environment, the code must be inspected and audited to guarantee that it does not contain harmful instructions.

This can be performed off-line in a trusted server that performs an executable verification, and signs the application to prove that it is harmless.

This technique allows the safe execution of application on a non modified environment, but requires the extensive study and analysis of the tasks code.

Sandboxing If it is impossible to guarantee the correctness of the downloaded code, it is necessary to isolate its execution. In the case of the execution of a malicious instruction, no harm should come to the host computer.

With sandboxing the downloaded code is executed in a restricted environment or virtual machine, not allowing harmful operations to interfere with the host operating system. The execution environment also intercepts the access to the host file system, preventing the access to system files. The execution environments can be either application level or system level virtual machines.

Interpreted language (such as Java, or .NET) execution environments fall in the application level virtual machines. Although these environments allow the execution of applications with full OS access, it is possible to define security policies to restrict access to some resources.

System-level virtual machines mimic a fully functional computer, and isolate the host operating systems. Any application executing inside such environments will see and execute within a fully functional guest operating system without affecting, modifying or compromised the host computer.

Adding to this advantage, the use of a virtual machine eases the development of portable code providing an uniform execution environment.

2.2.5 Analysis

In the following table we analyze 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: 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.

2.3 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.

2.3.1 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.

Task Tasks are the smallest work unit in a cycle sharing system. Each one is composed of the code to be executed and the data.

Although, some systems allow tasks to spawn themselves in order to split their work among their children, in most systems tasks are simple entities: the code is usually single threaded, running on a single processor, and tasks do not interact with each other (neither for synchronization nor message passing). IN these simple cases tasks have similar code, but process different data.

When submitting tasks, users have to define what code will be executed on the donor and the data that will be processed. The methods to define the code and the data differ between systems as will be seen later in this section.

Job Tasks executed to solve a common problem are grouped in jobs. These tasks, belonging to the same job, execute similar code, but have different input parameters.

A job must encase all the code necessary for the tasks execution: initialization code, task launch, tasks' code, and result retrieving. Furthermore, when creating a job, the user must define the input data for each task.

When defining a job the user must supply the code that will be executed by each task, and how the overall data will be split among different tasks. The creation of tasks can be automatically handled by the middleware, or programmed by the user.

Project Some jobs share the same code, only differing on the data sets processed. Without a higher level entity every user always has to submit the same processing code.

A project can be seen as a job template, where a user only defines the processing code. Later the project is instantiated to create a job.

In systems with the concept of project, when creating jobs the user only has to suply the data set to be processed and tasks' input parameters. The job's code has been previously defined when creating the project.

2.3.2 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.

Donor A donor is the owner of a computer where tasks from others are executed. The user must install the necessary software to execute tasks, and configure it. Before the execution of other users' work, the donor must select either the server where tasks will come from or select what kind of work he is willing to execute.

Job Creator On the other end we have users that submit work to be executed. In the simplest case a user is just a job creator.

Using the tools provided by the system, these users only submit the information about the job: tasks code and input data. If the system supports projects they just need to select the project and prepare the data.

Project Creator If a system support projects, someone has to create them: develop the processing code, and register it in the system.

If there are tools for the easy creation of projects, then any user can be a project creator. Later, the same user (or others) can create jobs that will be executed within the same project.

2.3.3 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.

(COTS) Executable Assignment If the system allows an external commodity off the shelf (COTS) executable assignment, to create a job the user only has to provide that application, either uploading it to the server or copying it to a suitable directory. In this step the user should also provide information about the executable: type of input/output data or the format of the data.

In the development of the executable, normally no use of special API is necessary. The application should only follow simple interface requirements imposed by the underlying system. Although a regular application can be used, to take advantage of some services (e.g. checkpointing) it may be necessary to use a supplied API, or link with special libraries.

In systems accepting executable assignment in the job definition step, the selected application just contains the code executed in every task. This application, being generic can not contain any task creation code, and is simply invoked on donor computers.

Module Development Some other systems require users to explicitly develop the processing code. In these systems the task's code should be encased in a module (or class if using object oriented languages).

The modules developed should comply with a pre-defined interface and can use a helper API in order to take advantage of available services.

The user is freed from developing task launching code, as this step is transparently handled by the systems.

Application Development Other systems require the explicit programming of the task launching step. Here, this step must be inside a complete application, that can also contain the data splitting among tasks and develop pre and post-processing steps.

A system that requires the complete application development must also supply the API for tasks launching and any other offered service. Using that, the user develops a complete application containing:

- environment initialization
- pre-processing code
- data partition code
- tasks' execution code (in a function or method)
- tasks' launch invocation
- result retrieval
- post-processing code

Although seting up a job this way is a complex feat, the set of problems solvable in such ways is more vast.

2.3.4 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

Data Definition The easiest mode to define a task is by simply performing data definition. The user only states (declaratively in a configuration file or with a user interface) what data each task is to process. These systems offer no auxiliary means to split the data, they just offer simple assignment mechanisms.

Data definition is used on systems where job creation relies on either executable assignment or module development. Then, the midleware just executes the executable (or module) on the donor computer, with the supplied input data.

Data Partition With data partition a script, or even the master code, splits the data that will later be implicitly assigned to tasks. The system offers both the means to split the data and to add the partial data to a pool. This addition can be performed programatically (by means of a API) or declaratively (trough a user interface or configuration file). In this case the underlying system is responsible for the creation and invocation of the various independent tasks, and to assign each task its arguments and input data.

Code Partition While with data partition, no explicit launch of tasks is performed, the user can have the possibility to control this launching. When job creation resorts to code partition, the user must explicitly launch each task using a specific API. The user must develop a complete application, as explained in the previous section, with all the necessary components.

As each task must process different data, the main program must also perform the data partition, and the data assignment (when creating a task).

2.3.5 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

Bag-of-tasks In a Bag-of-Tasks job all tasks are created simultaneously and execute the same code.

The main process only splits the data, creates the tasks, waits for their conclusion, and performs minimum processing of the results. Being the simplest parallel programming model, this reduces the programming effort to create such jobs. The user only has to develop the processing code , and rely on the middleware for launching each task and execute that code.

Further data processing (of the input and results) must be performed off-line with external tools.

Master-slave In a master-slave execution model, the main process creates the various task in a computational loop and blocks its execution until the end of every one of them. The tasks may be different between them (in terms of input data and code), started oat different times and having data dependencies between them.

In this kind of computation, there is a main program from where all tasks are launched, The user by programming the main function, can create simple Bag-of-Tasks jobs, or more complex workflows: the results of computations can be used as input to other posterior tasks.

Recursive In a master-slave computation only the main program can create tasks. Some problems require that running tasks can themselves create and launch others. Naturally recursive problems fit here, as well as problems where tasks also generate data to be processed.

In a recursive execution, the main application creates the first level of tasks (as in master-slave). These running tasks can also create other tasks when needed.

The programmer must develop the whole application: the main program, data splitting, and the tasks. In this model task creation is explicit, requiring the use of a supplied API.

2.3.6 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.

Launching Infrastructure For Bag-of-Tasks problems the simplest launching mechanism is by means of a launch infrastructure. The user supplies the application or module corresponding to tasks, defines the data to be processed, and the middleware becomes responsible for launching the necessary tasks.

The use of a launching infrastructure fits well and is the obvious solution to: i) Bag-of-Task problems, ii) task creation based on data definition, and iii) job creation based on executable assignment and module development.

Even when the application development job creation model is used a launching infrastructure is handy. In these cases the middleware configuration can be performed programatically (inside the main application) or declaratively (outside the application and through a user interface). In such cases, a launching infrastructure can be used to define the job application, and configure the middleware.

Monitor Infrastructure Independently of the way jobs are created and tasks launched, a status monitor infrastructure can be provided. The user can monitor the execution of the job and the status of the running tasks by means of a user interface. These monitor infrastructures, interacts with the middleware to provide a global view of the systems: available resources, and status of the submitted work. The status of the submitted work and can include the following: tasks not yet launched, tasks currently executing, terminated tasks, and results received (validated, note validated or corrupted). based on this information the user can easily decide to abort executing jobs, or predict termination times.

Programming API In order to take advantage of some offered services, it is necessary to use some programming API, for instance, when programming the tasks, an API can be offered to provide checkpointing.

When the user needs to develop a complete application, the use of API to define tasks is fundamental. These API are needed to register the data to be processed by each task or to explicitly create the tasks.

The existence of a API for code development is orthogonal to the existence of any kind of infrastructure.

Existing API are normally used to launch new tasks, control their termination, or provide check pointing to the code.

2.3.7 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.

Declarative When using a declarative task creation paradigm, the user can only state the input data of each task, either using a supplied user interface or by creating a configuration file.

This is mostly applicable when the user only needs to make an executable assignment and data definition when creating jobs and tasks.

Compiled To develop the processing code it is necessary to use an interpreted or compiled language.

The use of a compiled language allows full flexibility on the development of tasks code. The systems can even provide some APIs so that tasks can use some offered services.

Due to difficulties in making compiled code mobile, systems that use compiled languages require tasks to be a complete application (with the main function) instead of a module. The middleware uploads one complete application to the donor computer and starts it there. This architectural characteristic limits theses languages to mostly Bag-of-Task problems.

In order to have tasks running on different architectures, it is required that the user compiles the developed code to those architectures.

Despite these limitations, languages targeting high performance computing can be used (Fortran, for instance) and the tasks execute at full speed (much faster than if using a interpreted language).

Interpreted The use of interpreted or JIT-ed languages overcomes the issues raised with compiled languages.

The development of tasks as modules is straightforward and the deployment of that code becomes practical. Most interpreted languages (Java, C#, python) allow introspection, mobility and dynamic code loading, easing the development of a Distributed Computing infrastructure. Even from a complete application it becomes easy to isolate tasks' code (usually a function or class), and upload and invoke it on remote donors.

The use of virtual machines as execution environments increases the portability of the code. The donors are only required to have the language execution environment installed, all required libraries can be uploaded along with the task.

The added ease of setting up a donor can also increase the number of users willing to donate cycles.

The only drawback of using an interpreted language is its possible lower execution speed, that can be overcome by the gains of having a large donor base and ease of development.

Graphical Graphical languages should be used in conjunction with a more traditional programming language (either interpreted or compiled). In a Distributed Computing systems, graphical languages can be used on the definition of jobs, in the following aspects:

- Task code selection
- Data splitting
- Workflow creation

This may increase the number of users submitting work, making these systems more visible to the common computer user.

2.3.8 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

Explicit The user is obliged to contact the server and make an explicit selection of the projects or jobs he will donate cycles to.

Even on systems where a server just serves a single well known project, the user, when connecting, is explicitly selecting a project.

This explicit selection can be in the form of choosing a URL (of the server or the project/job) or by means of a user interface. The explicit selection of the project requires that the identity of the user creating the job or the focus of the work to be disclosed to the donors.

Implicit With an implicit job selection scheme, the donor does not have the means or possibility to select the jobs he/she will execute. The donor just connects to a server hosting various jobs, or to the peer-to-peer network, and the selection of the work is made by the system.

Restricted (topic based) In a intermediate level, users may be able to choose the subjects of the work they are willing to execute. In this case, the donor defines interest topics or a set of keywords allowing the servers (or the peer-to-peer network) to select the work to deliver to each donor.

This type of work selection does not require the donor to know the identity of the work creator, nor the precise purpose of the work, in order to match tasks with donors.

2.3.9 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.

User Ranking The easiest incentive to implement is user rankings. The ordering of donors depends on the quantity of donated resources to the community. When the donor correctly finishes a task the systems assigns points to it. To add a sense of accomplishment and boost the competitiveness between users, a ranking of the more meritorious users is publicly posted.

Processing Time On the other hand, after successful completion of a task, the donor can be rewarded with the right to use the system to execute his tasks (by means of processing time).

Later when scheduling tasks to be executed, the system should be able to use the rewarded execution time in a fair manner.

Currency In a similar way, a virtual currency can be awarded after the completion of a task. In systems with a market oriented scheduling policy, the collected values can be used to buy the processing time.

The awarded value can be used to bid for the latter execution of tasks on remote computers.

2.3.10 Analysis

The way the various systems implement and handle the user interaction issues is presented in two distinct tables: Table 3 (specifying the various possible user roles) and Table 4 (presenting programming and usage alternatives).

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.

	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

Table 3 (Continues on next page)

	Work organization	User Roles	Job creation Model	Task creation Model	Task execution Model
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 3: User roles

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	-	-	-	-
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 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.

3 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 [58]) 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 [43]. 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.

4 Conclusion

In this document we presented a taxonomy to evaluate and describe the various characteristics of Distributed Computing platforms. This taxonomy is split in three different classes of aspects: i) Architecture, ii) Security and reliability, and iii) User interaction.

This taxonomy was also applied to the Distributed Computing systems developed up to now, presenting the design decisions taken by each system.

Although all presented characteristics are relevant to the global and local performance of the system, some of those are fundamental for the system success in gathering users: i) Execution host security, ii) Network topology, iii) Scheduling policies, iv) User roles, v) Project Selection, and vi) Donation incentives. For each characteristic, we present the optimal and most efficient solution, those that would make a system successful and widely used.

Finally we compare these optimal solutions with those implemented in BOINC and conclude that they are not implemented in this system. The great success of BOINC does not come from the use of optimal solutions but from the use of two natural human reactions: empathy with the problem being solved, and competitiveness among users.

References

[1] S. Choi, H. Kim, E. Byun, and C. Hwang, "A taxonomy of desktop grid systems focusing on scheduling," Tech. Rep. KU-CSE-2006-1120-01, Department of Computer Science and Engeering , Korea University, November 2006.

[2] S. Choi, H. Kim, E. Byun, M. Baik, S. Kim, C. Park, and C. Hwang, "Characterizing and classifying desktop grid," *Cluster Computing and the Grid, 2007. CCGRID 2007. Seventh IEEE International Symposium on*, pp. 743–748, May 2007.

[3] S. Choi, R. Buyya, H. Kim, E. Byun, M. Baik, J. Gil, and C. Park, "A taxonomy of desktop grids and its mapping to state-of-the-art systems," Tech. Rep. GRIDS-TR-2008-3, Grid Computing and Distributed Systems Laboratory, The University of Melbourne, Australia, February 2008.

[4] M. Lobosco, C. L. de Amorim, and O. Loques, "Java for high-performance network-based computing: a survey," *Concurrency and Computation: Practice and Experience*, vol. 14, no. 1, pp. 1–31, 2002.

[5] K. Vanthournout, G. Deconinck, and R. Belmans, "A taxonomy for resource discovery," *Personal Ubiquitous Comput.*, vol. 9, no. 2, pp. 81–89, 2005.

[6] 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.

[7] 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.*, vol. 30, no. 8, pp. 207–216, 1995.

[8] 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*, (New York, NY, USA), pp. 165–172, ACM, 1996.

[9] 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*, pp. 181–188, ACM, 1996.

[10] P. A. Gray and V. S. Sunderam, "Icet: Distributed computing and java," *Concurrency - Practice and Experience*, vol. 9, no. 11, pp. 1161–1167, 1997.

[11] 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*, pp. 100–106, IEEE Computer Society, 1997.

[12] A. Baratloo, M. Karaul, Z. M. Kedem, and P. Wijckoff, "Charlotte: metacomputing on the web," *Future Gener. Comput. Syst.*, vol. 15, no. 5-6, pp. 559–570, 1999.

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

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

[15] A. Baratloo, M. Karaul, H. Karl, and Z. M. Kedem, "An infrastructure for network computing with Java applets," *Concurrency: Practice and Experience*, vol. 10, no. 11–13, pp. 1029–1041, 1998.

[16] H. Pedroso, L. M. Silva, and J. G. Silva, "Web-based metacomputing with JET," *Concurrency: Practice and Experience*, vol. 9, no. 11, pp. 1169–1173, 1997.

[17] L. M. Silva, H. Pedroso, and J. G. Silva, "The design of jet: A java library for embarrassingly parallel applications," in *Parallel programming and Java: WoTUG-20* (A. Bakkers, ed.), pp. 210–228, IOS Press, 1997.

[18] 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*, vol. 9, pp. 1139–1160, 1997.

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

[20] 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*, pp. 171–180, ACM, 1999.

[21] 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.

[22] Y. Amir, B. Awerbuch, and R. S. Borgstrom, "The java market: Transforming the internet into a metacomputer," tech. rep., Johns Hopkins University, 1998.

[23] 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*, (New York, NY, USA), pp. 140–147, ACM, October 1998.

[24] 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*, p. 592, IEEE Computer Society, 1998.

[25] L. F. G. Sarmenta and S. Hirano, "Bayanihan: building and studying Web-based volunteer computing systems using Java," *Future Generation Computer Systems*, vol. 15, no. 5–6, pp. 675–686, 1999.

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

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

[28] 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*, pp. 91–101, Springer-Verlag, 2000.

[29] 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*, pp. 582–587, 2001.

[30] 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*, pp. 1–12, Springer-Verlag, 2002.

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

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

[33] P. Cappello and D. Mourloukos, "Cx: A scalable, robust network for parallel computing," *Scientific Programming*, vol. 10, no. 2, pp. 159–171, 2002.

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

[35] 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.

[36] 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.

[37] 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*, vol. 1, pp. 259–266, May 2005.

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

[39] R. Gupta and V. Sekhri, "Compup2p: An architecture for internet computing using peer-to-peer networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 17, no. 11, pp. 1306–1320, 2006.

[40] 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*, pp. 346–349, ACM, 2005.

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

[42] 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.

[43] 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*, pp. 196–203, IEEE Computer Society, 2005.

[44] 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.

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

[46] 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*, pp. 49–55, IEEE Computer Society, September 2006.

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

[48] U. of Leiden, "Leiden classical." http://boinc.gorlaeus.net/.

[49] 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.

[50] 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*, vol. 0, pp. 248–253, 2008.

[51] 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*, pp. 1–8, April 2010.

[52] 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.

[53] 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*, pp. 20–24, 1999.

[54] 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*, pp. 341–348, 2000.

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

[56] R. Buyya, D. Abramson, and S. Venugopal, "The Grid Economy," *Proceedings of the IEEE*, vol. 93, no. 3, pp. 698–714, 2005.

[57] D. Lucking-Reiley, "Vickrey auctions in practice: From nineteenth-century philately to twenty-first-century e-commerce," *Journal of Economic Perspectives*, vol. 14, pp. 183–192, Summer 2000.

[58] B. Schmidt, "A survey of desktop grid applications for e-science," *International Journal of Web and Grid Services*, vol. 3, no. 3, pp. 354–368, 2007.