ISCTE IUL
Instituto Universitário de Lisboa

Instituto Universitário de Lisboa (ISCTE-IUL), Lisboa, Portugal

Conillon: Distributed Computing Platform for Desktop Grids

Hélio Alexandre Dias da Silva

hadsa@iscte.pt

A Dissertation presented in partial fulfillment of the Requirements for the Degree of Master in Open Source Software

Supervisors

Ph. D. Sancho Oliveira

Instituto Universitário de Lisboa (ISCTE-IUL), Lisboa, Portugal

Ph. D. Anders Christensen

Instituto Universitário de Lisboa (ISCTE-IUL), Lisboa, Portugal

ISCTE - IUL, Junho 2011

Instituto Universitário de Lisboa (ISCTE-IUL), Lisboa, Portugal

Conillon: Distributed Computing Platform for Desktop Grids

Hélio Alexandre Dias da Silva

hadsa@iscte.pt

A Dissertation presented in partial fulfillment of the Requirements for the Degree of Master in Open Source Software

Supervisors

Ph. D. Sancho Oliveira

Instituto Universitário de Lisboa (ISCTE-IUL), Lisboa, Portugal

Ph. D. Anders Christensen

Instituto Universitário de Lisboa (ISCTE-IUL), Lisboa, Portugal

ISCTE - IUL, Junho 2011

Agradecimentos

Exprimo os meus agradecimentos a todos os que, de certa forma, me ajudaram a concretizar este objectivo.

Aos meus pais pelo apoio e eterna ajuda que sempre me demonstraram, devo-lhes um especial e grande agradecimento. Sem eles este objectivo não teria o mesmo valor e propósito.

De maneira especial, agradeço à minha companheira e melhor amiga Patrícia, pela paciência e sentido de companheirismo demonstrados ao longo do desenvolvimento deste objectivo.

Ao orientador Sancho Oliveira, quero dizer que leva o trabalho de orientador bastante a sério, o apoio foi, desde o ínicio da tese, constante e teve sempre excelentes sugestões e com um acompanhamento firme na evolução deste trabalho e de todos os elementos que constituiram esta tese. Manifesto-lhe a minha maior gratidão por me ter conduzido nesta experiência divertida e enriquecedora. Ao co-orientador Anders Christensen quero dizer que o seu espírito crítico foi uma das mais valias na discussão de resultados e das suas sugestões teóricas para os artigos. Devo um grande obrigado a esta, excelente e profissional, equipa.

A todos o meu sincero agradecimento pelo importante papel que desempenharam na concretização de um objectivo a que me propus com todo o esforço e dedicação.

“A creative man is motivated by the desire to achieve, not by the desire to beat others.” -
Ayn Rand

Abstract

Grid computing enables organizations to integrate and share sets of heterogeneous resources into one synergetic and powerful system, in order to deliver efficiency and processing performance to demanding parallel applications. The aggregation and the coordination of these resources need to be integrated in a set of tools and protocols that reduce their complexity. Resource allocation is performed by a centralized manager that has complete knowledge of system state.

Conillon is a distributed computing platform that manages and coordinates a set of actors. This platform enables heterogeneous workstations to function as part of a distributed system. Conillon is a lightweight, simple to use, and yet allows any number of different applications to be deployed and executed in parallel. Our platform maximizes the resource utilization in benefit of the demanding parallel applications. Parallel applications developers have access to a set of tools that simplifies the development process and the creation of tasks. The complexity related to the scheduling, distribution, and fault tolerant management of tasks is shielded from the programmer. Conillom platform has an integrated set of programms that work synergistically delivering a positivo ratio between efficiency and processing performance.

We present our platform in detail as well as the results of extensive experiments about performance. The platform behaved relatively well and achieves an average speed up of up to x1.9 on the time needed to complete a job each time the computational resources are doubled.

Keywords: Distributed computing, Parallel computing, High Performance Computing, Fault tolerance, Performance, Java

Resumo

A computação paralela permite integrar e partilhar um conjunto de recursos heterogéneos num sistema poderoso e sinergético, de forma a poder disponibilizar eficiência e desempenho de processamento às aplicações paralelas. A agregação e coordenação destes recursos devem integrados num conjunto de ferramentas e protocolos que reduzam a complexidade inerente. A alocação de recursos é realizada por um gestor centralizado que tem conhecimento de todo o estado do sistema.

Foi desenvolvida uma plataforma de computação distribuída, o Conillon, que coordena e gere um conjunto de participantes. A nossa plataforma permite que estações de trabalho heterogéneas façam parte de um sistema distribuído. A plataforma Conillon é simples, ocupando poucos recursos no entanto permite que múltiplas e distintas aplicações sejam executadas em paralelo. A plataforma maximiza a utilização de recursos em benefício das exigentes aplicações paralelas. Os programadores de aplicações paralelas têm acesso a um conjunto de ferramentas simples para criação de tarefas. A complexidade relacionada com o escalonamento, distribuição e tolerância a falhas é encapsulada do programador. A plataforma Conillon tem um conjunto integrado de programas que funcionam sinergicamente, entregando um rácio positivo entre eficiência e performance de computação.

Neste trabalho presentamos a nossa plataforma em detalhe, bem como os resultados de testes extensivos sobre performance. A plataforma comporta-se relativamente bem, alcançando uma aceleração de até 1.9x no tempo necessário para terminar um serviço sempre que o número de recursos computacionais é duplicado.

Palavras-Chave: Computação distribuída, Computação paralela, Computação de alta performance, Tolerância a falhas, Performance, Java

Contents

List of Figures

List of Algorithms

List of Tables

1 Introduction

Scientist and engineers are building more and more complex applications, to process this large data sets and execute scientific experiments, massive computational resources are required, in order to meet this demand the computing power had to change the computing paradigm. This paradigm is known as distributed computing, and it is present in many areas where high computing power is key factor in obtaining value.

Nowadays, this important factor is extremely decisive in areas such as drug design, weather prevision, ecological modeling, operation research, search for extraterrestrial intelligence, among others, [1, 7]. This paradigm is so critical that the scientific community asks, sometime, for help from volunteers for these demanding computing processes. This request for help, became possible because the Internet connections started to have sufficient bandwidth to transfer the necessary applications, and more importantly, current processors are true processing machines. However, with the evolution of computing, GPUs have been presenting strong arguments in support of volunteer computing and large-scale computing.

Distributed computing presents an added value for science, and for those who need to compute large amounts of data within a reasonable amount of time. Despite the constant evolution of hardware, a single machine may not have enough computing power to compute a massive volume of parallel data. We cannot forget the price that a super computer can have, a high performance machine, with special features for the purpose it was intended, has a high price. Here arises the problem of companies with insufficient resources to enter into this kind of systems. There has been an increasing interest in utilizing existing workstations for distributed computation when they would otherwise be idle (such as outside office hours, during breaks and so on). Depending on the organization, a workstation may be idle for a significant part of the time, normally 95% [36].

1.1 Motivation

Over the years the technology development presented an abrupt progress, this contributed to the increase in network bandwidth, processing power via CPU or GPU, and also the storage capacity. This combined with a system that allows the interconnection of various systems, heterogeneous or not, makes it possible to build a Grid of computers that can be used in parallel and distributed computations. Typically, these computations are intensive, complex and require system capabilities beyond what a single machine could not in good response time. Many applications can benefit from such approach [13]. A parallel system can be easily built with home computers and networking equipment. Such systems have a much lower cost than a super computer. This process is called cluster, which is defined by a set of interconnected computers, dedicated to a common resource. However, specific software is required to maximize these resources, the parallel distribution of tasks and the use of processing units must be maximized. This software optimization is called grid computing. Conillon has born from the need to create a low cost Grid computing system for small and medium enterprises. This aims to link workstations sub harnessed to contribute to the required computation locally. Conillon is designed to be opportunistic in obtaining resources. Conillon is multi-platform and is developed in Java. Although this system is designed to work in LANs, it also offered the possibility to interconnect workers via

WAN, this feature allows external users to provide their computing resources of the institution. This platform was designed to offer developers easy development of parallel applications without having to worry about issues of synchronization and distribution of tasks among the available resources. Thus, it is possible to develop a wide range of parallel applications with the same programming model.

1.2 Objective

This work aims to develop the following objectives:

- Research and develop a distributed computing platform;
- Scalable - dynamic addition of nodes
- Maximize the use of the CPU;
- Fault-tolerant engine;
- Ability to cope with multiple problems simultaneously;
- Easy to use and manage;
- Support for different programming languages;
- Maximize performance and efficiency;
- User accessibility:
    - Multi-platform;
    - Applet - ability to use the web browser to explore the CPU cycles;
    - Screen-saver application - to use in the different Operative Systems;

1.3 Outline of this Thesis

This document begins by introducing the background needed to understand this thesis, also it is presented some of the different distributed computing platforms, in Chapter 2.

In Chapter 3, it is described Conillon architecture and all the involved actors in detail. We discuss the Coordinator, the Code Server, the Client and the Workers, these are all the actors that take part of our platform. We focus on the details of each actor. The chapter is based in all the development work done during the completion of the thesis. Parts of this chapter have been published in [31] and [33].

Chapter 4, describes the Conillon programming model, in here all the information needed to develop a parallel application using the Conillon platform is presented. An simple example is presented and explained.

In Chapter 5, we present the developed parallel applications used to evaluate Conillon. Various experiments have been performed to show how the platform behaves in a single and multiple applications running. Parts of this chapter have been published in [32].

Finally, in Chapter 6, conclusions are drawn and it is presented the achieved goals, furthermore it is also presented a new area where this platform could benefit as future work.

2 Background and Related Work

Currently there are numerous distributed computing platforms, some of them have been playing an important role in the scientific community. I will present the necessary background to understand this thesis and the most important platforms in the context of this work. We also give an overview about grid computing, existent topologies, security and advantages.

2.1 Background

Grid computing belongs to class of distributed systems. Andrew Tanenbaum defines a distributed system as:

> "A collection of independent computers that appears to its users as a single coherent system." [35]

A good analogy is presented by Foster and Kesselman:

> "The word Grid is used by analogy with the electric power grid, which provides pervasive access to electricity and has had a dramatic impact on human capabilities and society. Many believe that by allowing all components of our information technology infrastructure-computational capabilities, databases, sensors, and people-to be shared flexibly as true collaborative tools the Grid will have a similar transforming effect, allowing new classes of applications to emerge."[13]

According to IBM's definition:

> "Grid computing, most simply stated, is distributed computing taken to the next evolutionary level. The goal is to create the illusion of a simple yet large and powerful self managing virtual computer out of a large collection of connected heterogeneous systems sharing various combinations of resources.“ [11]

Grid computing provides access to advanced heterogeneous computational resources to scientific community. Aims sharing and aggregation of distributed resources, based on the cost, performance and on the organization processing requirements [20].

Grid computing can explore underutilized resources, a batch job that needs significant amount of resources is perhaps the ideal use for a grid. Explore the parallel CPU capacity is one of the most attractive features of a grid, a CPU intensive grid application can be divided in small jobs and executed on different CPU resources. For example, a good scalable operation, finish twice as fast than if it uses twice the number of processors, see Chapter 5.2, for more information about performance. However, not all applications can be transformed to run on a grid, application designers must identify and explore parallelizable algorithms for the developed application. Another important grid computing contribution is the collaboration of other identities, for example, organizations can share resources collectively as a larger grid, CERN [1] explores this feature [30]. Grids can access to additional resources, CPU's are not the only necessary resource, storage resources are also important.

[1]CERN European Organization for Nuclear Research, is a centre for scientific research. Visit www.cern.ch, for more information about CERN

Grids are organized in three topologies, as illustrated in Figure 1, the simplest topology is the intragrid, which is limited to single organization. This organization can be made up of number of computers and network devices and is relatively easy to share data between nodes. The extragrid is based on the intragrids, expanding the fact by connecting two or more intragrids. This topology involves a more complex management and security cares. The intergrid is the biggest topology and is explored by scientific community, financial industry, pharmaceuticals industry, among others. This grid requires integration of various organizations, customers and resources.

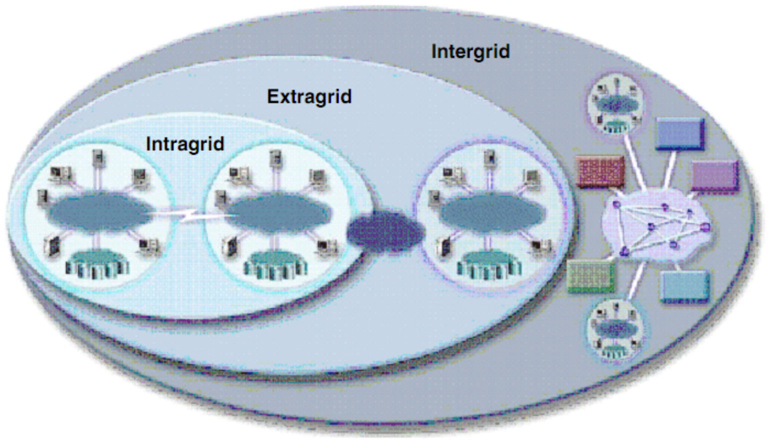


Figure 1: Intragrids, extragrids, and intergrids. Image obtained from [11]

Grids offer inexpensive reliability, management software that can automatically submit jobs to different resources when a failure is detected.

Security is an important factor in planning and maintaining a grid. Heterogeneous resources can potentially be infected with virus or other malicious software, for this reason, it is necessary understand which grid resources need to be secure. Firewalls[2] can add an additional layer of protection from internal or external users, by using firewalls it is possible to limit the network communications and use only allowed protocols and ports.

Shortly after Java was released in 1995, a number of middleware project for the distribution of Java programs sprung up. Javelin was released in 1997 and it was one of the first implementations of a parallel computing platform for Java. In 2000, version 2.0 of Javelin [9] was released. SuperWeb [2] is a parallel computing infrastructure developed in Java proposed in 1997. In 1999, Bayanihan [28] was proposed and it was developed using Java. As the internet became increasingly popular towards the end of the 1990s and as more computers got connected, Seti@HOME[5] was released. Seti@HOME allows volunteers to contribute computational resources to the analysis of cosmic radio data in the search for extraterrestrial intelligence. The Berkeley Open Infrastructure for Network Computing, BOINC [3] grew out of the Seti@HOME project and allows for other computationally expensive tasks to use Seti@HOME's distribution model. The Condor project was created at Wisconsin University. Condor [37] is a high-throughput distributed batch computing system and can be considered one of

[2]Firewall - Service designed to prevent unauthorized access between two or more networks. It is considered a first line of defense in protecting data or devices.

the first grid systems. Like Condor, InteGrade [14] is a middleware that allows for scheduling and monitoring of parallel applications.

More recently, there has been research into distributed computation based on a peer-to-peer model: XtremWeb [8] is a multi-user, multi-application for global computing and grid computing research. XtremWeb aims at turning a set of volatile resources spread over a network into a runtime environment executing highly parallel applications. P3 [24] (which stands for Parallel Peer-to-Peer) is an Internet computing platform designed for high-performance fault-tolerant parallel computing and is developed in Java.

2.2 Javelin

Javelin [21, 9] is a computing platform designed to link participants with internet or intranet connection to the Javelin. This system, allows the volunteer to log into a web browser and participate in the Javelin computation using the Java Applets. While developing the Javelin, the creators had in mind the simplicity in participation, it just needs a web browser with Java Applet and the extension service URL. Javelin is divided in three types of entities: brokers, clients and hosts. Client is a process that needs computer resources, host is a process that offers CPU cycles and needs a web browser with Java Applet enabled to participate, broker is a process that coordinates the supply and demand for computing resources. For the host participate in the computation it requires opening the corresponding URL. Clients, can create jobs by developing the corresponding applets and host the applets at the broker. The broker is a HTTP server that makes the accommodation of applets developed by clients and makes the distribution of tasks to hosts.

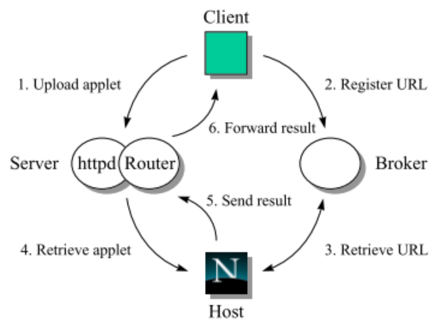


Figure 2: Javelin architecture. Image obtained from [9]

As illustrated in Figure 2, Javelin needs some procedures to be able to start exploring resources. The client needs to upload applet that has developed to an HTTP server. Then, it needs to register the corresponding URL in the Broker. The host makes the request for tasks and receives the corresponding URL and it will start downloading the applet developed by the client. Upon completion applet will run. The result is then send to the server. The server receives the result.

2.3 Bayanihan

Bayanihan [28] is a project that aims to explore the idea of voluntary distributed computing . This project uses the benefits offered by the Java applet using a web browser to explore the capabilities of volunteers resources. Since this is a distributed computing system, some problems and challenges of parallel computing had to be solved, such as fault tolerance, performance, scalability and performance, parallelism, and accessibility. In order to outcome these problems, Bayanihan uses Java technology and HORB. HORB [29] is an extension to Java, that distributes objects across the network, offering portability and interoperability between applications.

Bayanihan is composed of two main components, the server and the worker clients. Worker clients can offer their CPU to perform the computation, but can also be watcher clients who only have access to results and statistics. Clients can be started by a command line application or Java applet, using the Web browser. Each client contains a motor that connects to the server in order to get the necessary data objects and start the execution of parallel problems. The server is an application that contains one or more parallel problems.

2.4 Seti@home

The SETI @ home [5] arose from the need to search for extraterrestrial intelligence using the resources provided by volunteers. Volunteers can participate in the distributed computing project, offering their CPU cycles for the development of SETI @ home. The computational model of this project is very simple, the data signals captured by the antenna is divided into work units of fixed size and are distributed to various clients via the Internet. Customers will then calculate a result and after the calculation will send the result to the server, the process is then repeated. This project is famous by the amount of volunteers who joined the project. The SETI @ home project uses the BOINC platform to explore the remote resources of volunteers.

2.5 BOINC

BOINC [3, 4] is a distributed computing platform with a component that exploits the resources of volunteers around the world. This is an open source project and is being developed at UC Berkeley. The main objective of the development of BOINC is to utilize the resources offered by volunteers. For this very specific goals included:

- Reduce barriers to entry for new clients, servers BOINC require little maintenance and project managers need to have moderate knowledge of the system.
- Sharing of resources among autonomous project, the projects are not centrally administered, each project has its own server, however volunteers can, if they wish to participate in various projects.
- Support for various applications, there are several mechanisms that allow to manage multiple BOINC projects, such as systems for scheduling and data distribution.
- Reward the volunteers, participants to offer their resources are offered credit. These credits reflect the total resources that the PC could contribute voluntarily to date.

Boinc creates a set of programs that aims to solve a problem. Each project has its own servers and the project developer is responsible to develop the necessary software to be send to the volunteers. Each project, has its work units. Work units are a set of data and parameters submitted by the developers and intended to be run at the volunteers computer.

Boinc architecture is very simple, see Figure 3, the server runs a data base that stores the registered users, the completed work units and available, and others. Each project has a service responsible to distribute and collect the work units and the results. Other service is responsible to distribute the necessary files and data and to collect them when necessary. The schedulers have access to the technical information of each resource and send work units by its technical characteristics. At the client side, there is a core that is common to all BOINC clients and the client code that is specific to the project.

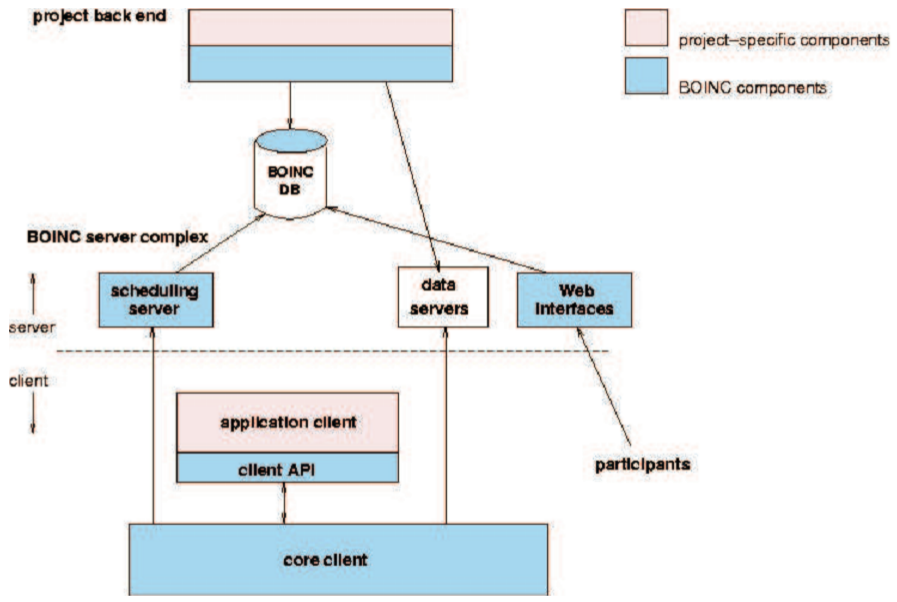


Figure 3: BOINC architecture. Image obtained from [39]

2.6 Condor

The Condor project [37] was created at Wisconsin University. Condor is a high-throughput distributed batch computing system. Condor can be considered one of the first grid systems and it was created in 1988 and it is used in academic and enterprise organizations.

Condor can be organized in several groups of computers, this groups are called Condor Pools and they can be present at the same administrative domain or in different locations of LAN. Inside the Condor system, exists several services responsible for the coordination of the grid. As illustrated in Figure 4, when a user submits a job to the Condor grid, an Agent (schedd) is responsible by saving the sobs in a persistent media while it tries to find resources that fit that job profile. Agents and resources have to communicate with central manager (Matchmaker), which makes an analysis to find compatible agents and resources. When the central manager gives this information the agent contacts the the Resource (startd) and both start a new process, inside a Sandbox, and the resource starts the job execution. To the job be successfully executed, a Shadow is responsible to provide all the necessary details to complete the application.

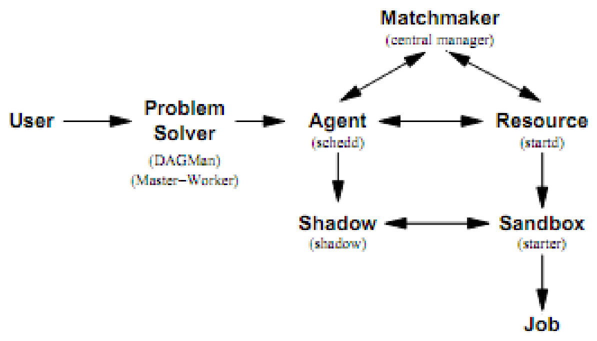


Figure 4: Condor major process - Image obtained from [37]

2.7 P3

P3 [24],which stands for Parallel Peer-to-Peer, is an Internet computing platform designed for high-performance fault-tolerant parallel computing developed in Java. P3 is organized in Nodes. Nodes are divided into two types: manager nodes and compute nodes. The manager nodes are responsibility for the coordination and maintaining quality of service of the P3 network state. This nodes must have permanent Internet connection and good overall performance. Manager nodes have the unique responsibility of controlling the peer routing, file-system and caching and maintain the application status.

The compute nodes contribute with disk storage and processor time to a single application. The work assigned to each compute node is dictated by the manager. P3 has built-in support for finding and storing Java objects by reference and it is in this way is possible to create an object-space. Objects are stored across the available nodes.

2.8 Summary

In this chapter, it was introduced all the necessary literature necessary to understand this thesis. Next, it was discussed the related work, several similar projects exists with different characteristics and limitations. All of them explore the idea of using CPU cycles to take advantage in parallel applications.

3 Conillon

Due to the heterogeneous nature of a grid, there can be some difficulty in develop different parallel applications and distribute them by a grid. Different operating systems, libraries and programming languages makes distributed computing a very complex work.

Conillon tries to be language neutral, and support a variety of programming languages that compile to JVM bytecode. Java presents advantages appropriate for the distributed computing field [40], code mobility, garbage collection, polymorphism, and can be used to allow interoperability between heterogeneous systems [10]. Furthermore, JVM allows to users develop their distributed applications in several different languages such as Scala [23], Jython [16], and JRuby[3].

3.1 The concept

Conillon is an intragrid distributed computing platform with extragrid capabilities, designed [31] to exploit available resources in a small or medium organizations. In the design of Conillon, we therefore prioritized the following aspects:

- Maximize CPU usage of idle desktop computers: Conillon should maximize the CPU utilization of the desktop computers that take part in the distributed computation. In order to achieve this goal, workers cache jobs so that they do not have to wait for a new job to be sent after they have submitted the result of a computation.
- Fault tolerance: Conillon should tolerate that workers dynamically become unavailable without warning. If a worker suddenly becomes unavailable, any jobs already sent to the worker should be delegated to another worker. Furthermore, if a worker is available but it found to be unreliable, (e.g., if the worker frequently disconnects from the system), Conillon should exclude the worker from any future computation.
- Extendable: In future versions, Conillon should allow for extensions such as allowing conditional scheduling of jobs that take advantage of the GPUs present on the workers and allow for the execution of native applications.
- Multi-purpose: Conillon should allow for several jobs of different type to be run at the same time. Conillon should be able to compute different tasks from different clients.
- Dynamic addition of new types of jobs: It should be possible to add new jobs (including both data and code) to a running system. Hence, Conillon must take care of distributing the both code and data on-the-fly.
- Lightweight and ease of use: Conillon should be simple to install and use. Applications developers should be able to easily take advantage of a dynamically changing pool of desktop computers.

[3]JRuby - http:// jruby.org/ for more information on JRuby

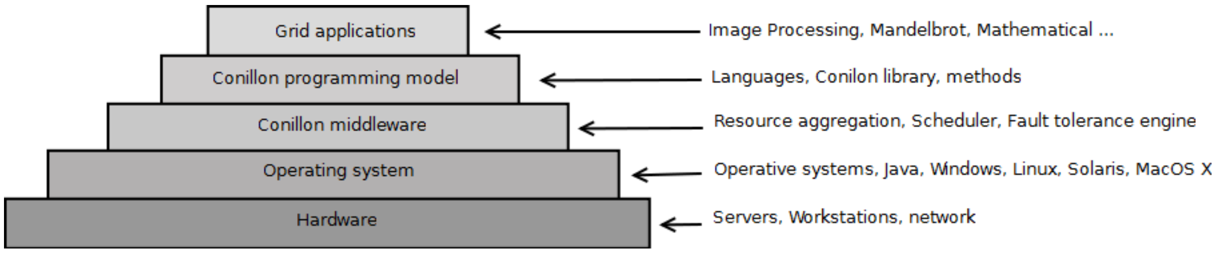

Grid applications
Conillon programming model
Conillon middleware
Operating system
Hardware
Image Processing, Mandelbrot, Mathematical ...
Languages, Conilon library, methods
Resource aggregation, Scheduler, Fault tolerance engine
Operative systems, Java, Windows, Linux, Solaris, MacOS X
Servers, Workstations, network

Figure 5: Conillon grid layer concept

Figure 5, depicts the logical hierarchical layers in which Conillon platform depends. At the base of the hierarchy is the *Hardware*, Conillon depends on a good Hardware infrastructure, servers, network equipment, and workstations that will serve as compute nodes. The *Operating system* layer, includes all the software needed to run the platform, operating system and a Java Virtual Machine is needed. Above this layer, the *Conillon middleware,* provides all the necessary services for resource aggregation, scheduling, fault tolerance engine, code distribution and task/result mechanism, to function. This layer hides complexity from developers. The *Conillon programming model* layer, includes the Conillon library and the developer tools needed to develop grid computing applications. The top level layer, the *Grid applications,* consists in the developed applications, in our case Clients, that need massive computational resources.

3.2 Architecture

Conillon is a distributed computing platform [33]. The design is based on the master-slave model[27]. Two servers, the Coordinator and the Code Server, act as masters. Both of the masters do the preprocessing service where they prepare the slave to run a specific task, and the postprocessing where the masters receive the result and deliver it to the right Client. The two servers are responsible for all the coordination and code distribution. There are two types of slaves in Conillon's architecture, Workers and Clients. Workers are computer resource donors that connect to the platform and donate resources by executing tasks. Clients are applications that require a large amount of computation. A client divides a computationally intensive job into a set of independent tasks that can be executed in parallel and submits those tasks to Conillon to be executed. An overview of Conillon's architecture is depicted in Figure 6.

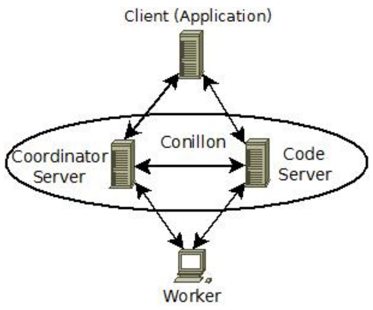


Figure 6: Simple Conillon architecture

The master-slave model was adopted because it presents several potential advantages over a peer-to-peer architecture like efficient scheduling, fast fault tolerance management and an easy process for adding/removing Workers and Clients to/from the platform [17, 26]. This model is also used to overcome NAT[38] limitations, if not configured, a router does not accept new connections from the WAN, this action has to be explicitly configured. With master-slave model, the slave, in our case the worker, always makes the first connection to the server, this policy is well accepted by the routers, cause the connection is made from inside to outside.

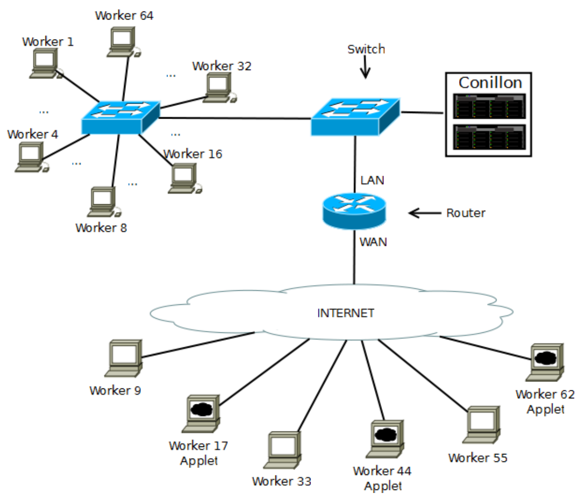


Figure 7: Network topology

Conillon platform uses multiple communications streams, these are done using TCP protocol[25], this protocol operates at Transport layer from the OSI model and offers reliability in the transmission, error detection, flow control and congestion control. Conillon trades commands and data, transporting them over the LAN/WAN, see Figure 7. TCP provides the required end-to-end reliability to the flow of communications.

The platform is designed to run on the Java Virtual Machine. Java. Conillon platform explores Java class loading mechanism, using the *ClassLoader*[4] object, this mechanism is responsible for locate or generate a definition for the class.

The general flow of data and code starts by the Client sending a set of tasks to the Coordinator. These tasks are then scheduled to run on the available free Workers. The Workers execute tasks and send the results back to the Coordinator. These results are immediately forward to the Client. The Code server provides the code necessary for loading and executing the tasks.

3.2.1 Task

Task is CPU intensive computation that is part of an application. In Conillon, tasks are send from the Clients to the Conillon platform and then dispatched to available resources. When the computation is calculated a result is delivered.

[4]ClassLoader - http://download.oracle.com/javase/1.4.2/docs/api/java/lang/ClassLoader.html for more information on Java *ClassLoder*

3.2.2 Result

Result is an outcome produced by a series of calculations defined in the task. A Task produces a Result. And the output of task must be implemented by the developer. The result is send from the available nodes to the Conillon platform, and then it is dispatched for the resource requester.

3.3 Coordinator

The Coordinator plays a central role in the Conillon architecture, as illustrated in Figure 8. The Coordinator manages all the interactions between the Clients and the Workers. This service is responsible for the task scheduling and distribution, fault management, maintaining worker status, and all task related operations.

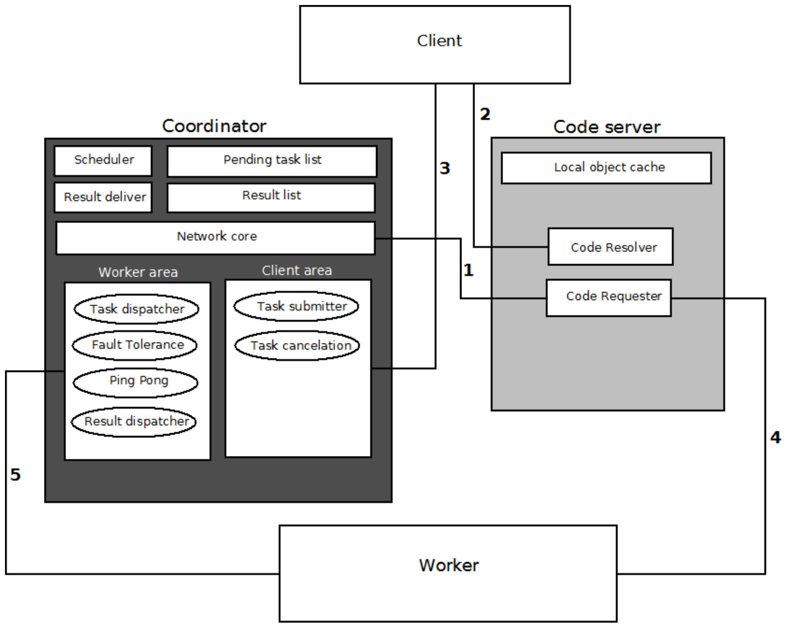


Figure 8: Coodinator Services - Task Delivery engine, Worker engine with fault tolerance, Client engine. The numbers indicate the initial sequence when the system is started, the lines represents a network connection.

The Coordinator scheduler sends new tasks to Workers as the Workers request them.

Clients can submit tasks with different priority levels. The priority levels can vary between *Low, Normal, High and Very High* and tasks are scheduled according to a fair round-robin scheduling algorithm. This means that *Low* priority tasks are also scheduled even if a large amount of Client jobs exists with higher priorities.

As can be seen in Figure 9, the scheduling algorithm begins by dispatching tasks from the first Client, the number of scheduled tasks varies with the Client priority. E.g. for a client with high priority tasks, 6 tasks are dispatched and, for a client with low priority tasks only one is dispatched. If more than one client is present the scheduler is always circulating the Clients lists and dispatching tasks based on their priority.

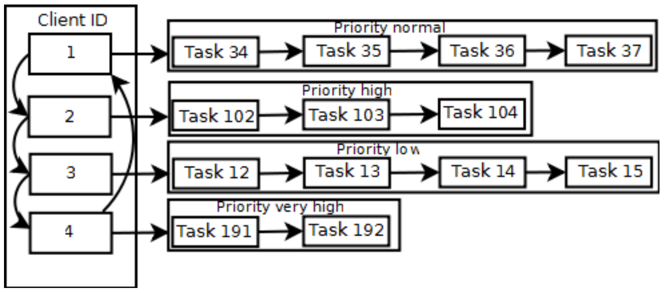


Figure 9: Round robin scheduling

Once the Coordinator receives a task, it's added to a list of pending tasks and is state is set to Schedule. After the task is sent to the Worker, its state is set to Send. If communication is lost to the Worker, the task status changes to Failed and the task is reinserted into the list of pending tasks in order to be rescheduled. Finally, when the result of a task is received and forward to the client, the task is marked as Completed and removed from the Coordinator. Tasks state transitions are shown in Figure 10

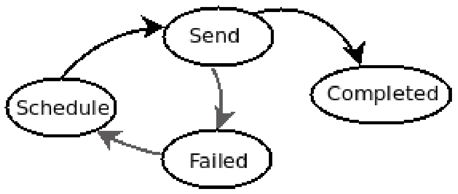


Figure 10: Tracing task status

The Coordinator keeps statistical information about the client tasks and about the performance of the Workers. This information is used to optimize future scheduling and allows the exclusion of unstable and otherwise problematic Workers. The information collected is also used to minimize the number of times that new code has to be sent to a Worker by scheduling the same type of tasks for the same Worker as often as possible.

A fault management module is responsible for detecting and handling malfunctioning Workers. The liveness of the Worker is tested using a "*ping pong*" protocol, this mechanism is illustrated in Figure 11. The Coordinator periodically sends a "*ping*" request to each Worker, and the Worker has to respond with a "*pong*" within a configurable amount of time. If a Worker does not reply, the Coordinator assumes that worker is not operational and removes it from the system. If the Worker did not respond due to a temporary communication problem, the Worker can re-join the set of compute nodes again.

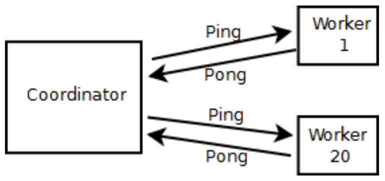


Figure 11: Conillon ping pong protocol

All tasks that were being calculated by a malfunctioning Worker are rescheduled and, eventually, sent to other Workers. With this simple approach, Conillon can handle all types of Worker issues that result in the absence of a "*pong*", such as abrupt termination of the Worker process, software bugs, hardware malfunctions, and network failures. The fault tolerance engine works via multithreaded system, if a problem is detected the Coordinator maintains the normal operation and the fault tolerance operations are kept running in parallel, saving the tasks scheduled to the pending task list.

For an efficient transfer of results to Clients, the Coordinator server has a multithreaded delivery engine that sends results to clients concurrently. With this architecture, the Coordinator can deal with several client applications at same time. The concurrent design of the task deliver system allows clients to receive the results in a timely manner.

Coordinator maintains a very strict operation regarding the Clients. If a network problem or an illegal behaviour is detected, with one of the Clients, all its tasks are immediately erased from the system, an alert is sent to the Workers and all the information about the Client is deleted. When the Workers received the alert, they will kill the running threads related to that Client. The Client can still rejoin later and needs to submit the full job again.

3.4 Code Server

Code server allows Conillon to be a flexible and dynamic platform with respect to the tasks run on the Workers. The Code server is responsible for distributing all the code needed for executing tasks. This server is the coordinator of the code distribution sub-system and provides a transparent mechanism to transfer objects (including bytecode) between the other parties of the platform. The first time a new type of task is sent from the Client to the Coordinator, the Coordinator does not know the bytecode associated to the task. The code distribution sub-system handles with this situation in a complete transparent manner to both the Client, to most modules of the Coordinator, and to the Workers. Since the required bytecode is new, the Coordinator and the Workers send request to the Code Server. The Code server fetches code, as needed, directly from the clients and stores the code in a local cache. Normally the code is associated to the needed classes.

The Code Server cache keeps control of the Client, if the Client leaves the system, all the Client related structures are deleted. When Code Server fetches code to the Clients, they will produce an array of bytes of the need class and send this array back to Code Server.

In this way, the Code Server becomes a resource server that hosts the necessary bytecode to run the tasks submitted by the clients. All of this is handled automatically by Conillon and the users do not need to specify what code needs to be uploaded to the Code Server. The necessary bytecode is stored in a HashMap. The key of this Map is a *ClassRequest* which stores information about the class, problem number, version and the Client. The value of the Map is the associated code necessary for the task execution and is stored in a array of bytes.

As illustrated in Figure 12, the flow of data and code is started by the Client that sends tasks to the Coordinator, see flow number **1**. In this case, Pi tasks[5], when the Coordinator loads the task there is no definition for the received class, Coordinator sends a request for the Code Server asking for the Pi task code, see flow number **2**, Code Server searches the cache for the requested definition, as the

[5]Pi task more information regarding Pi task can be found in Chapter 5.1.1

cache does not contain the requested code it is necessary to request the task code from the Pi Client, see flow number **3.** The Client sends the requested code to the Code Server, see flow number **4**. Code Server will save the code in the cache, and send the code to the Coordinator, see flow number **5**, future request for the Pi task code will be directly answered by the Code Server, no need for the flow **3** and **4**. The Coordinator can now define and creates a Pi Task object and start scheduling Pi Tasks. At the Worker side, the process is the same, when the Task is received and no definition exists at JVM the code is requested to the Code Server. In this case the code for P72 Task already exists at Code Server's cache.

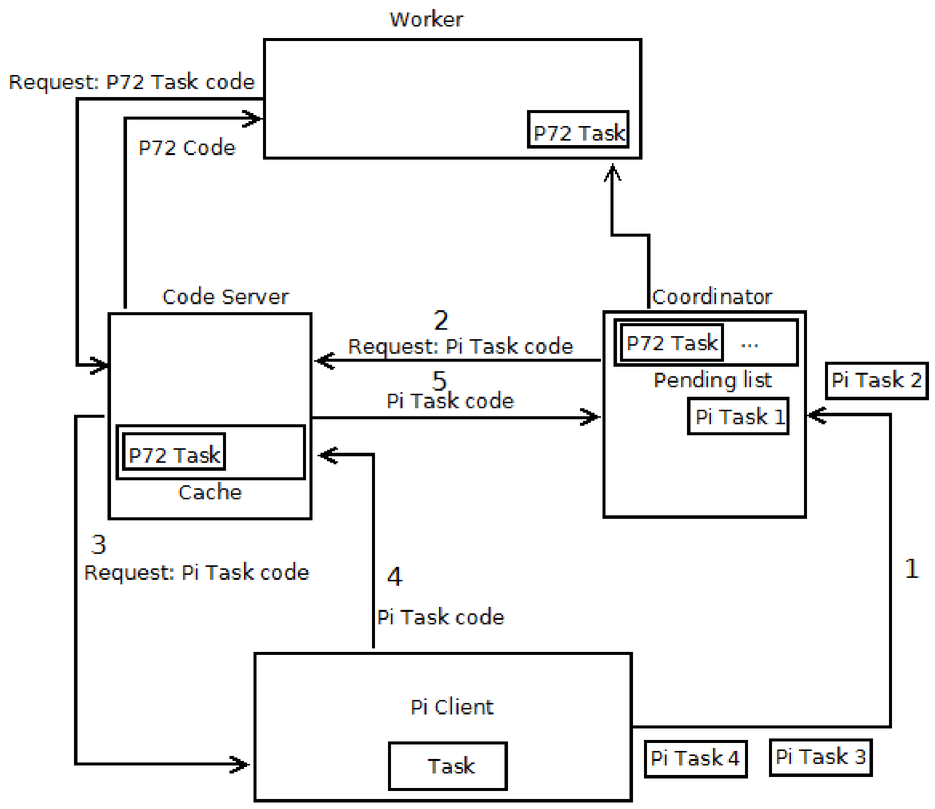


Figure 12: Code flow from all actors. The numbers indicate the order by the flow of code is traded.

3.4.1 Web Server

Web server is a module added to Conillon to serve HTTP[12] requests to the Java Applets. Due to security reasons, Java Applets cannot request code directly to the code server so Conillon dynamic class loading mechanism cannot be used, this way it was necessary to create a web server module that serves the necessary classes to the Java Applets via HTTP protocol. This module listens for connections using the 8080 port and parses the information send from the requester, in this case the Java Applet. The requester will send an GET request for the necessary data to the Web server, the web server will respond, sending the requested data via HTTP protocol.

As can be seen in Figure 13, when a Worker tries to load a task the Applet will send a GET message to the Web server containing the name of the needed class, the Web server module will query the Code server for the code and then dispatch the code back to the requester. This way, is possible to explore workers resources using the web browser and maintain the Conillon dynamic code distribution.

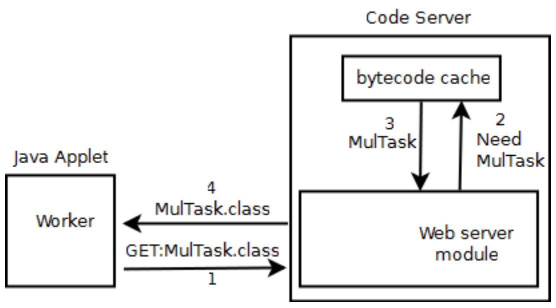


Figure 13: Code Server module: Web server

3.5 Worker

A Worker is a process that runs on each compute node, designed to fully explore the available resource. Workers are heterogeneous, each one has its own physical and virtual features. The application is designed to explore the heterogeneous capabilities of each worker.

The Worker is divided into two parts, a core zone and a working zone. The working zone is a sandbox[6] that provides a safe execution area. The sandbox protects the core and the compute node from eventual task problems and illegal behavior, while still providing task with the conditions necessary for performing the required computation.

When the sandbox loads a task for execution, there is no class definition present, a Code Sever request is done with the necessary class by the core. The class is send in array of bytes and loaded with the *defineClass()* method that converts an array of bytes into an instance. The class loaded can then be defined and created.

The core handles all the communications with the Coordinator and the Code Server, and it is responsible for requesting the tasks and the necessary data. The Worker process starts by detecting the number of CPU cores. Knowledge about the number of CPU cores is necessary in order to fully exploit the computation resources of the Worker. The number of concurrent tasks executed on a Worker corresponds to the number of detected CPU cores.

A cache mechanism has been developed. To reduce the overhead of the communication and to maximize CPU usage, tasks are cached. This caching mechanism allows for tasks to be run back-to-back. In a 1 Gbit/s network, we have experienced up to a 10% increase in performance on a Worker with four cores when a cache of up to four tasks was used.

For this experiment we used a workstation running Ubuntu 9.10 with an Intel i7 950 - 3.06 GHz processor, this is a four core processor plus *Hyper-Threading* [19], so it is detected by having 8 cores. To run this experiment it was used the Pi Client, see Section 5.1.1 for more information about Pi Client, with 10 000 decimals and a work size of 50, where 5 runs of each cache size were done. The cache size studied were 0, 4 and 8, where 0 represents no cache, 4 represents half of the available detected cores

[6]Sandbox - Provides an detached area for applications to run in, protecting the rest of the machine against malfunction applications in the sandbox.

and 8 represents caching the same number of cores available. Figure 14 shows the average time each cache size gained. The total time needed to complete the job with no cache is 154.6s, while for the test with cache size 4, the time was 139,8s this represents a major gain of 10%, meanwhile, the cache size 8 produced no gains, getting a time of 139.9s.

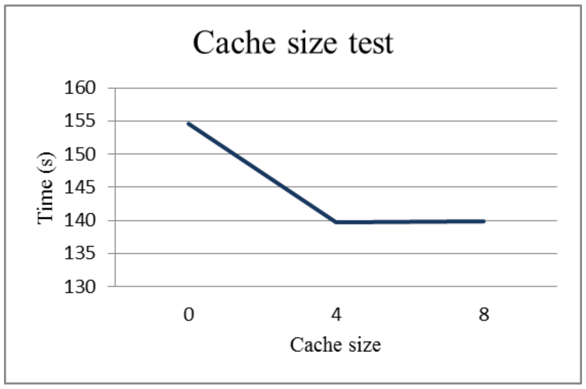


Figure 14: Cache size test

Cache size	0 (No cache)	4 (50% of CPU cores)	8 (100% CPU cores)
Average time (s)	154.6	139.8	139.9

Table 1: Average obtained times from the cache experiment

When there is free space on the cache, the Worker automatically requests one or more tasks from the Coordinator this mechanism is illustrated in Figure 15. When the tasks are loaded to the sandbox, the system verifies if the necessary code already exists at the JVM system, if not the Worker will request the necessary classes to the Code Server.

If a Client suffers a abnormal behavior, e.g. Disconnected due to network problems, or a flag is send from the Client to cancel all tasks, the Coordinator sends a message to all running Workers to kill the threads running the specific client tasks. Workers will kill the running threads related to that worker. When the thread is killed, and available slots exist, the Worker requests more tasks to the Coordinator and starts the task computation.

Worker maintains a very controlled behavior about the running and completed tasks. The metrics are sent to the Coordinator and then to the Client. This way the Client can produce several statistical outputs about the running computation.

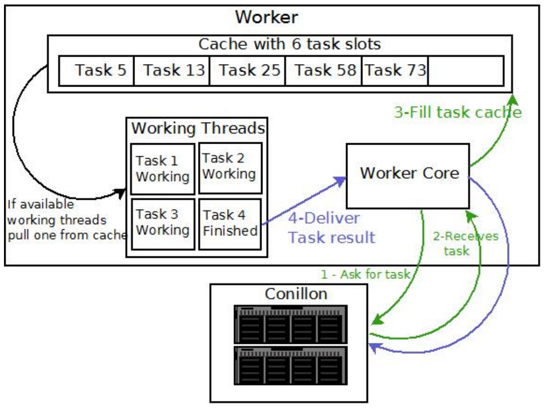


Figure 15: Worker asking for classes and dispatching the results.

Resources can be explored in various ways, a user can install and run a simple application that gives him a simple screen saver and thus, the user, can start offering CPU cycles to the grid. Meanwhile, if the user offers are sporadic, he can start the web browser and point to Conillon URL, a Java Applet with a worker application will be loaded and ready to exploit available resources.

3.5.1 Worker Screen saver

Worker application has its own screen saver. We designed a simple and low CPU consumption screen saver that draws random squares with random size on the screen along with the displaying the number of processed tasks by the Worker, see Figure 16.

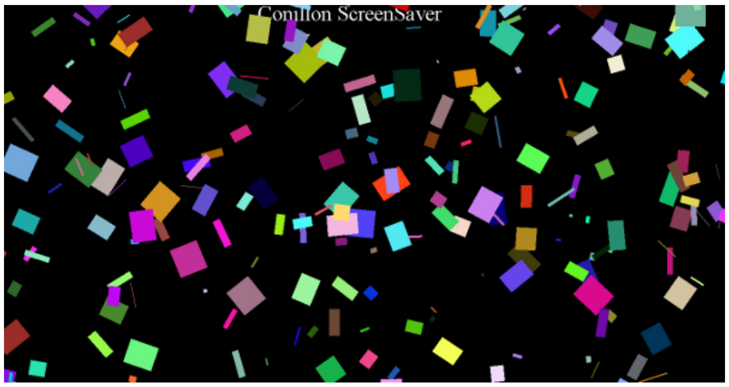


Figure 16: Conillon Screen saver

3.5.2 Worker data

Several attributes are necessary to control a worker. This attributes are necessary to produce statistics and various control information and are known by the Coordinator and the Worker. This object is created when a worker connects to Conillon and stores all the flow produced by the Worker, as illustrated in Figure 17, all the timings related to task calculation is stored, as well as, the tasks processed, starting and finishing times, number of processors available by that worker, the operating system and the IP address.

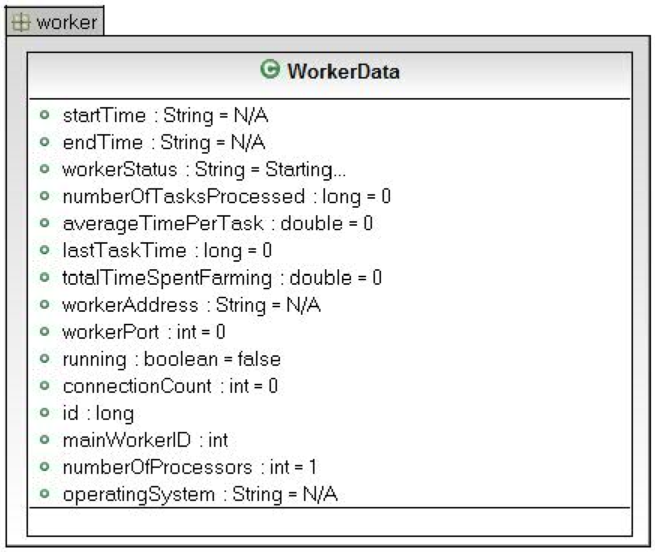

worker
WorkerData
startTime : String = N/A
endTime : String = N/A
workerStatus : String = Starting...
numberOfTasksProcessed : long = 0
averageTimePerTask : double = 0
lastTaskTime : long = 0
totalTimeSpentFarming : double = 0
workerAddress : String = N/A
workerPort : int = 0
running : boolean = false
connectionCount : int = 0
id : long
mainWorkerID : int
numberOfProcessors : int = 1
operatingSystem : String = N/A

Figure 17: Class *WorkerData* - Worker attributes used to identify a compute node

3.6 Client

A Client is a program written by a Conillon user who needs access to a large amount of computational resources. The user should program the client so that the computational job is divided into a set of independent tasks, it is his responsibility to identify the parallel code portions that can take advantage of distributed computing. As can be seen at Figure 18, tasks are submitted to the Coordinator who distributes the tasks among the available Workers.

There are two other services in the Client, the Code Provider and the Results Retriever. The Code Provider is responsible for answering requests from the Code server. The Code Provider service and the Code server hide all the complexities of the code transfer. This approach allows users to connect new client applications to the platform transparently. All issues related with new code transfers are done automatically and this feature combined with the Code server give Conillon the capacity to execute tasks submitted by several different Clients at the same time. If the Client uses threads to send Tasks to the Coordinator, it is possible to receive Results at same time, Conillon is designed to maximize performance and minimize waiting times.

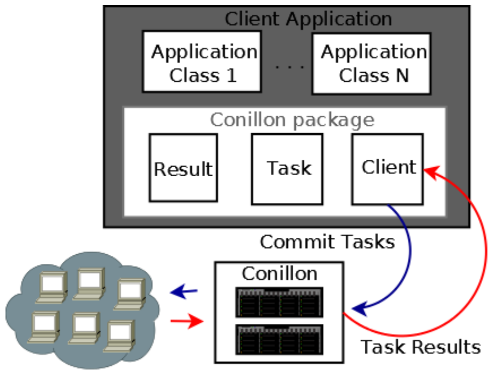


Figure 18: Client interacting with Conillon servers - Submitting and receiving tasks

The interaction between the application developer and the platform is done via extending the class *Client.* This class has access to all necessary mechanisms to send and receive data between the Client and the Conillon platform. As illustrated in Figure 19, the developer has access to six important methods: *commit(Task task), commitAndWait, cancelAllTasks(), startWork(), disconnect()* and *GetNextResult().* The method *commit()* and *commitTaskAndWait()*, are responsible to send the created task to the Coordinator, but while the *commit()* method tells to the Coordinator that the task is ready to be processed, the *commitAndWait()* orders Coordinator to wait for the developers order to start the process, to give the start order use the *startWork()* method. If a problem is detected, and the developer wants to cancel all tasks, *cancelAllTasks()* method will send a flag to the Coordinator containing an id to cancel all tasks from that specific Client, read more about the Worker's behavior regarding this process in Chapter 3.5. When the developers wants to start receiving the results, *getNextResult()* method will query a list for results, if no results are available it will wait, if results are available it will remove the first in the list, and returns the corresponding object.

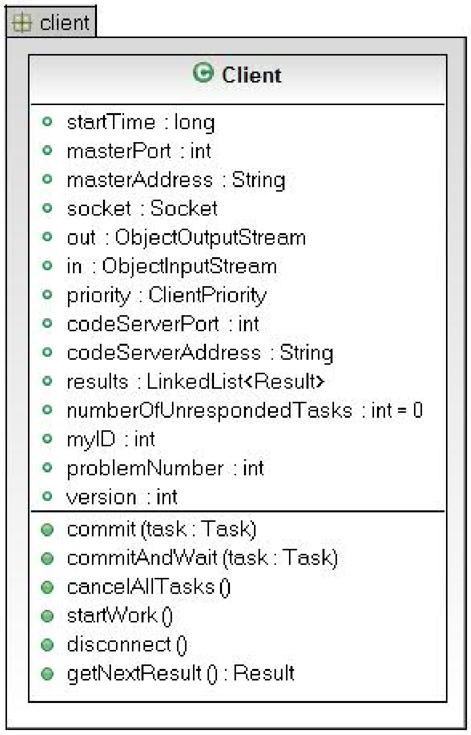


Figure 19: Class diagram - *Client* class

3.6.1 Client data

ClientData is an object that identifies a connected user that needs computational resources. This object is created when a client connects to Conillon. This information is used to produce statistics about the client. As can be seen in Figure 20, the Coordinator knows how many tasks have been injected and how many have been delivered. The IP address and the id are simple control information. The problemNumber and version are explained at Chapter 4.3.

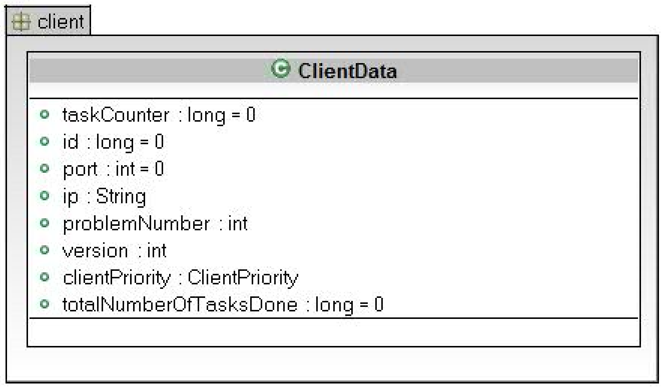


Figure 20: Class *ClientData*: Client attributes used to identify a user that needs large amount of resources

3.7 Security

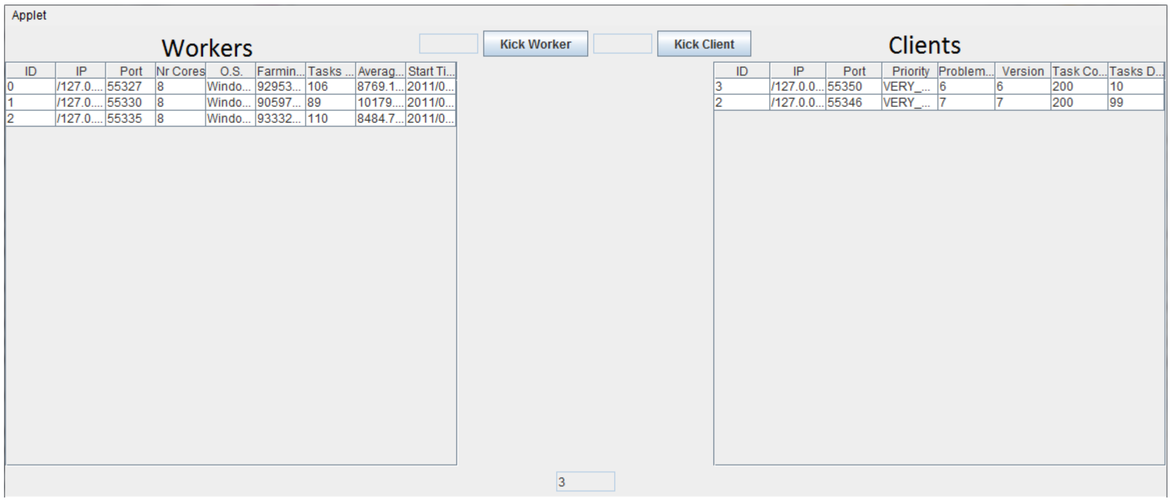
Security is an important issue to take in account, if the grid computing system is compromised, can potential be used has an massive attack tool. Java as some security mechanisms that addresses some security problems, others is the developer's responsibility to identify and secure. Worker Applet has a double protection mechanism, the first is the JVM security policy, applets that tries to access files from the host machine is checked before access to the requested resource, this policy is not allowed by default. The second is the sandbox created at the Worker core, this sandbox creates a safe execution area, where it loads code in a separated thread, if the thread suffers an abnormal termination the core maintains intact.

The second mechanism above mentioned is present at the Worker application, however if a malicious client injects tasks with invasive code, the worker will execute the code with no restrictions. For this problem to be solved a Java policy must be created, protecting the host resources from being access from the Worker application. Still, Java offers a simple protection at bytecode level, the JVM ensures that illegal operations are blocked.

3.8 Conillon administrator

Administering distributed computing platforms can be complex. Conillon design ensures that the management is self-sufficient with little user administration.

Conillon administrator is a small applet created to administrate clients and workers connected to the system. The applet shows several information about workers, IP address, number of cores, average time per task, and others. For the clients it's possible to observe the running application (version and problem number), total number of tasks and delivered tasks. This information is parsed from a list of objects (*WorkerData* and *ClientData* above mentioned) send from the Coordinator. The administrator offers the possibility to disconnect clients and workers.



Workers

ID	IP	Port	Nr Cores	O.S.	Farmin...	Tasks ...	Averag...	Start Ti...
0	/127.0....	55327	8	Windo...	92953...	106	8769.1...	2011/0...
1	/127.0....	55330	8	Windo...	90597...	89	10179....	2011/0...
2	/127.0....	55335	8	Windo...	93332...	110	8484.7...	2011/0...

Clients

ID	IP	Port	Priority	Problem...	Version	Task Co...	Tasks D...
3	/127.0.0...	55350	VERY_...	6	6	200	10
2	/127.0.0...	55346	VERY_...	7	7	200	99

Figure 21: Conillon administration panel

3.9 Summary

Conillon platform is divided in two central services in which the Clients and the Workers trade data. The Coordinator and Code Server are designed to be dynamic and to allow opportunistic exploration of resources. Coordinator does the task distribution and deliver part and Code Server sends the necessary code to the requester.

Workers can participate in the computing voluntary using the web browser or the worker application. Worker application has a built in screen saver that can be configured to exploit CPU resources when the computer is idle. The experiment showed a benefit of 10% using a task cache.

Client can easily interact with the platform, only a couple of methods are necessary and all the dynamic distribution of code is already implemented.

In resume, the Conillon model, previously defined, as the following flow:

1. Clients send tasks to the Coordinator
2. Coordinator schedules the tasks to the available Workers
3. Requester's ask for *bytecode* to the Code Server
4. Workers do the active calculation and send the Result to the Coordinator
5. Coordinator sends the Results to the right Client
6. Client processes the Result

4 Conillon programming model

The programming model of Conillon was design to provide a simple interface between the developer's application and the middleware. All the complexities related to distributing tasks, collecting results, ensuring fault-tolerance and so on, are hidden from the programmer.

4.1 Task

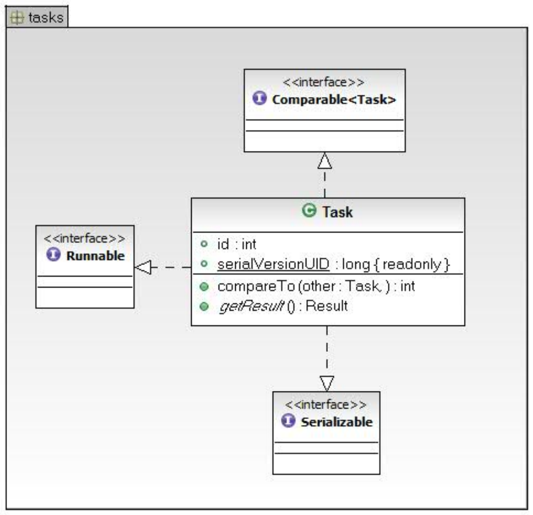


Figure 22: Class diagram - *Task*

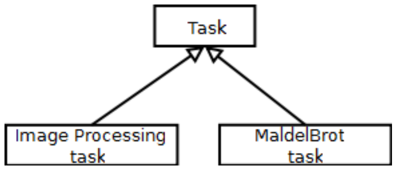


Figure 23: Abstract class *Task* and two subclasses: *ImageProcessing* task and *Mandelbrot* task

Task, as can be seen at Figure 22, is defined has an abstract class with one abstract method *getResult()*, explained at Chapter 4.3, and implements the *Runnable* class, this implementation is due to the fact that tasks are designed to be executed in threads. The *Runnable* interface imposes the *run()* method, and is necessary to lunch a thread in the Worker sandbox. *Task* also implements the *Serializable* class, this interface is used to save the object state and to transport the object through the platform. This class is subclassed by the Clients and it must implement the necessary algorithms that take part of parallel application. An example is shown at Figure 23, where 2 different classes inherit the *Task* class, although the subclasses are different, for the platform, this difference is unnoticed, the task will be executed using the *run()* method.

4.2 Result

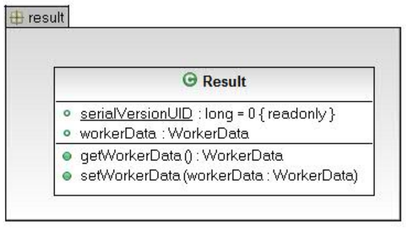

result
Result
serialVersionUID : long = 0 { readonly }
workerData : WorkerData
getWorkerData () : WorkerData
setWorkerData (workerData : WorkerData)

Figure 24: Class diagram - *Result*

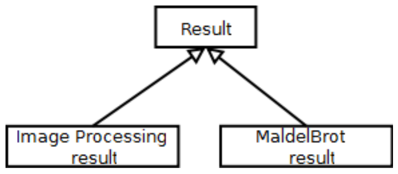


Figure 25: Abstract class *Result* and two subclasses: *ImageProcessing* result and *Mandelbrot* result

Result, just like *Task*, is an abstract class, but with no method declaration. The class *Result*, as can be seen at Figure 24, has access to all data related with the Worker, this can be used by the application developer to produce statistical information, more about *WorkerData* can be consulted at Chapter 3.5.2. This class is subclassed by the developers and it must implement the necessary representative object called the result, an example is illustrated in Figure 25, where two different results inherit the *Result* class, for the platform these objects are merely *Results* no matter the implementation.

4.3 Model

The two main parts of the interaction between the client application and Conillon are: committing tasks to Conillon and retrieving the results once the tasks have been computed. Hence, the development of a new client application is simple and straightforward. In order to develop an application to run on Conillon, the user adds the Conillon library to his or her project. This library has three main classes which are: the *Client* class, the *Task* class, and the *Result* class. The *Client* class, see Figure 19, provides methods that take care of the interaction between the client application and the rest of the Conillon platform, and needs a set of mandatory information: (*ClientPriority* priority, *int* problemNumber, *int* version, *String* coordinatorAddress, *int* coordinatorPort, *String* codeServerAddress, *int* codeServerPort). This information is necessary to define the Client's priority, the Coordinator and Code Server address and port. The application version and problem number, are for the developer keep track of send versions.

The main methods of the *Client* class are:

- *commit(Task task)* - sends the specified task to the platform (that is, to a Coordinator). When the task has been sent, the method returns.
- *getNextResult()* - waits for the first available result produced by a task that was executed on a worker. Tasks are sent to the worker in the order they were received by the Coordinator, but, since workers are different, the results can become available in any order.

The client task must extend the *Task* class. This abstract class has one method that needs to be implemented:

- *run()* - this method is executed by the worker and returns a result. The *Result* object, which needs to be created in order to save the task calculation.

- *getResult()* - this method returns the calculated *Result* object, and is used by the Client to retrieve the task output.

The client application should also define a result that must extend the Result class. The result class is where developers can define a specific task result. Both Task and Result subclasses should be programmed according to the application's requirements. Read the next Chapter for an example of an simple distributed application.

4.4 Implementation

Bellow is described a simple parallel job that multiplies two numbers in distributed way and then sums all the multiplied numbers at the client. This program is to be intend has a simple implementation of a parallel job in the Conillon system. The objective of this application is to multiply two numbers in distributed way, so the Task has to multiply two numbers, see Algorithm 2. The result of the multiplied number must be defined, the *MulResult* class will store the result of the multiplied numbers, see Algorithm 1. As you can see in the Algorithm 3, the tasks are going to be submitted to the Coordinator and the results are going to be summed.

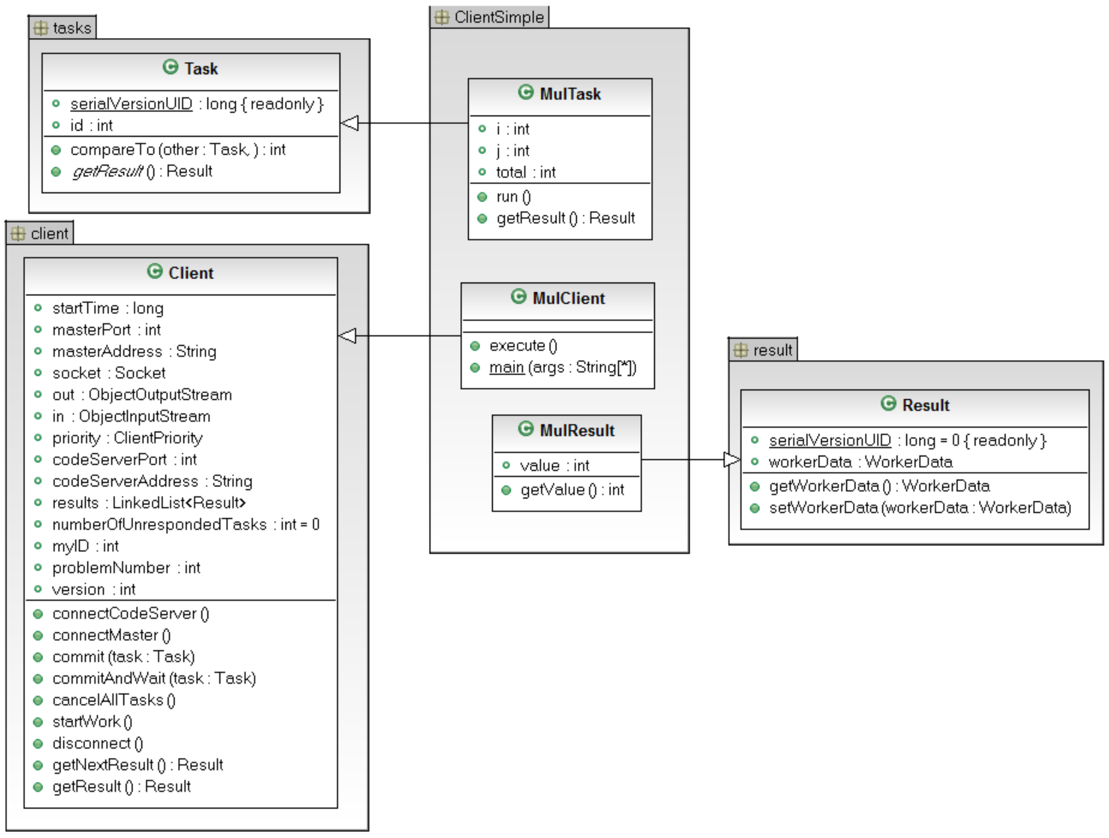


Figure 26: Class diagram of the simple client and considering the inheritance

Algorithm 1 *MulResult* Class

```
import result.Result;
public class MulResult extends Result {
        private int value;
        public MulResult(int value) {
                this.value = value;
        }
        public int getValue() {
                return value;
        }
}
```

First, as you can observe in Algorithm 1, it is necessary to create a object representing the Result. The result is the final representation of the task calculus. This means whatever the task calculates the result will represent the task output. In this example we will multiply two Integer numbers, so the result can be an Integer. The method *getValue()* is used by the Client to retrieve the calculated result.

Algorithm 2 *MulTask* Class

```
import result.Result;
import tasks.Task;
public class MulTask extends Task {
        private int i;
        private int j;
        private int total;
        public MulTask(int i, int j) {
                super(i);
                this.i = i;
                this.j = j;
        }
        @Override
        public void run() {
                total = i *j;
        }
        @Override
        public Result getResult() {
                return new MulResult(total);
        }
}
```

Second, as you can see at Algorithm 2, it is required to create the Task. The task will multiply two Integer numbers (i , j). The calculus will be done at *run()* method. This method is executed by the Worker and will produce a result and is saved with the *MulResult* class.

Algorithm 3 *MulClient* Class

```
import client.Client;
import comm.ClientPriority;
public class MulClient extends Client {
        public MulClient(ClientPriority priority, int problemNumber, int
            version, String masterAddress, int masterPort, String
            codeServerAddress, int codeServerPort) {
                super(priority, problemNumber, version, masterAddress,
                    masterPort, codeServerAddress, codeServerPort);
        }
        public void execute() {
                int numTasks = 1000;
                long total = 0;
                for (int i = 0; i < numTasks; i++) {
                        commit(new MulTask(i,i));
                }

                for (int j = 0; j < numTasks; j++) {
                        total += ((MulResult) (getNextResult())).getValue();
                }
        }
        public static void main(String[] args) {
                new MulClient(ClientPriority.VERY_HIGH, 8,8, "localhost", 0, "
                    localhost", 0).execute();
        }
}
```

Third, as you can observe at Algorithm3, it´s a requisite to submit the created tasks to the Coordinator. The tasks are submitted using the *commit(Task)* method. This method will send the created tasks to the Coordinator. When the Client wants to receive the Results, it needs to use the *getNextResult()* method, that will retrieve the result produced by the Worker and sent from the Coordinator, this result is stored in a list at Client side. The task committing and receiving can be accelerated using threads, depending how intensive tasks are and the number of available Workers, Conillon system can dispatch results without having the full task list submitted. When the *MulClient* is created, there are necessary inputs that must be defined, explained in Chapter 4.3.

4.5 Summary

To develop a distributed computing application using Conillon developer´s only have to create two classes, a Task and a Result, then it´s necessary to interact with Conillon server using several given methods. There way is easy for application developers implement embarrassingly parallel applications without worrying about distribution and/or synchronization issues.

5 Developed applications and performance evaluation

Some Clients have been developed in the context of this work. In this section is presented the created clients and the experiments done.

5.1 Developed applications

The developed applications have the programming model based in the Chapter 4.3. All the applications use inheritance from the following abstract classes, *Task*, *Result* and *Client*. In this Chapter is presented the four different applications, with distinct task and purposes, sharing the same programming model.

5.1.1 Pi

The Pi Client calculates theπ number using the Bailey-Borwein-Plouffe Pi Algorithm [6]. This formula provides an algorithm for the computation of the nth binary digit of π. The formula is:

$$\pi = \sum_{i=0}^{\infty} \frac{1}{16^i}\left(\frac{4}{8i+1} - \frac{2}{8i+4} - \frac{1}{8i+5} - \frac{1}{8i+6}\right).$$

Figure 27: Formula used to calculate the nth digit of π. Image obtained from [6]

Client Pi is divided in three classes, *PiClient*, *PiTask* and *MyResult.* In the *PiTask* class it was implemented the Bailey-Borwein-Plouffe Pi Algorithm has a task using the *run()* method, this way, it's possible to create tasks that calculates different positions of the Pi number. The *MyResult* class is the production of the task calculus, in this case, to store a big number, it was used the *BigDecimal.* The *PiClient* class has access to the necessary methods to interact with the platform, this class crests injects the tasks and receives the results calculated by the workers and produces the final Pi number.

Example of the first ten and last ten digits of a 10 000 digits π: 3.141592653 ... 3877582375.

The source code of this Client can be checked on Chapter 8.1.

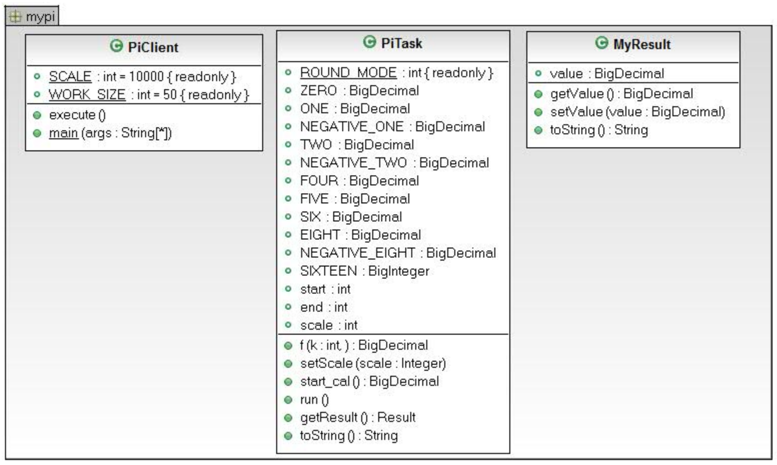


Figure 28: Class diagram for the Pi client

5.1.2 MandelBrot

The Mandelbrot [18] Client calculates a representation of the Mandelbrot set as an 8000x8000 pixel image. This image, see Figure 29, is set of points that produces distinctive fractal shape.

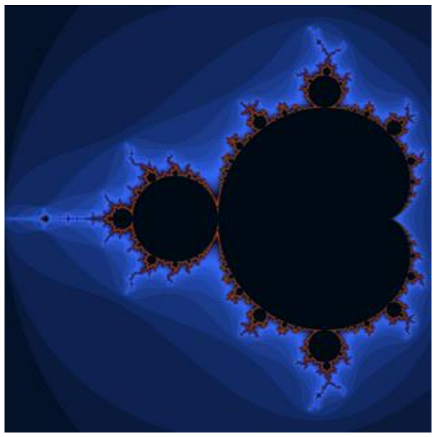


Figure 29: Mandelbrot representation

This application is divided in three classes, *MandelbrotClient*, *MandelbrotTask*, *MandelbrotResult.* The computation of the image is divided into partial images of 200x200 pixels each, this produces a total of 1600 tasks, the *MandelbrotTask* class calculates each partial image by calculating the set of points of each portion, this calculation produces an image saved in array of bytes, this array of bytes is the result of each portion of the image, each portion is saved in the *MandelbrotResult.* The MandelbrotClient sets the maximum number of iterations normally, 4096 or 8192 in the computation of each pixel in the Mandelbrot image. This class has access to the necessary methods to interact with the platform, this way, it creates and injects the tasks, a total of 1600 tasks, and receives the image portions. Once computed, the partial images are compressed in JPEG format and sent back to the Client where the complete image is assembled.

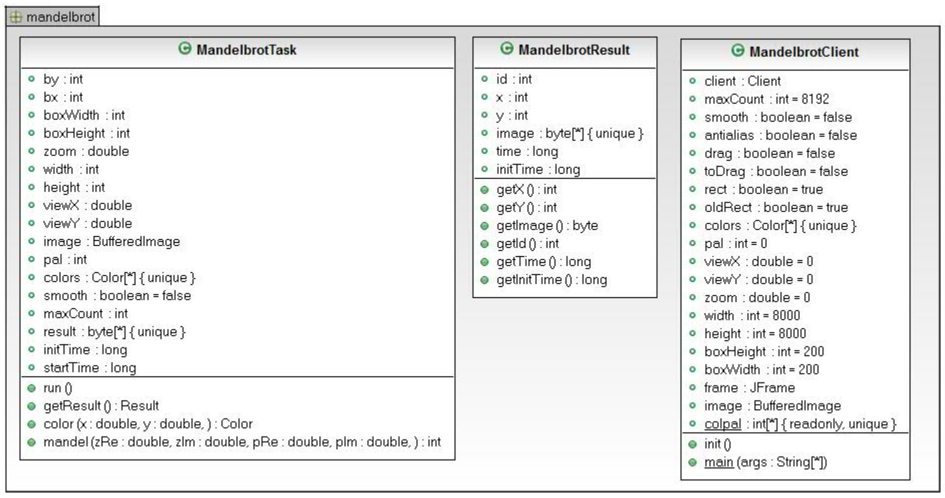


Figure 30: Class diagram for the Mandelbrot client

The source code of this Client can be checked on Chapter 8.3.

5.1.3 Image Processing

The image processing Client applies a blur filter to 1000 high definition (1920x1080) images of a rendered video. Each frame is compressed in JPEG format to a size of around 75 KB.

Image Processing application is divided in three classes, *ImageProcessingClient*, *ImageProcessing-Task* and *ImageProcessingResult*. The image processing job is divided into 1000 tasks (one task for each image), where the *ImageProcessingTask* applies the blur filter using a kernel of size 8x8 pixels to the image. The applied filter produces an array of bytes has a result, this result is stored in the *ImageProcessingResult* class. The *ImageProcessingClient*, loads all the 1000 high definition image and creates the tasks, each task is loaded with an image. The created tasks are injected to the platform. The received results are the images with the applied filter.

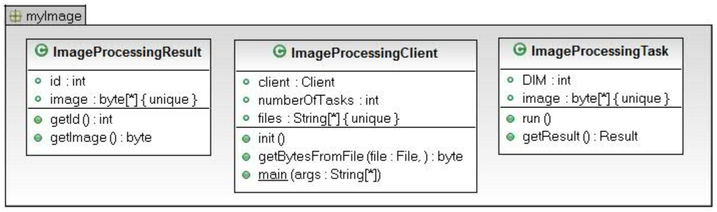


Figure 31: Class diagram for the Image processing client

The source code of this Client can be checked on Chapter 8.2.

5.1.4 P72

The P72 Client calculates the greatest common denominator between two numbers from a given interval using the Euclidean [15] method. This application follows the model discussed in the Chapter

4, divided in three classes, *ClientP72*, *TaskP72* and *ResultP72.* The *TaskP72,* calculates how many greatest common denominators between two numbers exists from a given interval, each interval is a task. The result is a regular *Int,* and represents the sum made in the task. The *ClientP72* class interacts with the platform injecting tasks and receiving the sums obtained by the tasks. This application can be configured with a given size of numbers and the interval for each task.

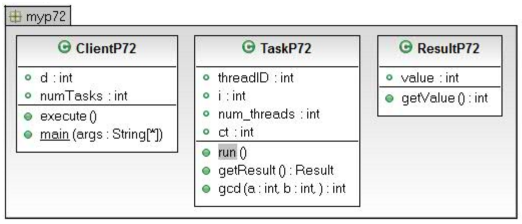


Figure 32: Class diagram for the P72 client

The source code of this Client can be checked on Chapter 8.4.

5.2 Experiments and results

5.2.1 Hardware and Software

The set of Workers used for the performance experiments was composed of 64 desktop computers. Each of these computers had an Intel Core 2 Duo 3.06 GHz (dual core processor) with 4 GB of RAM and was running Windows XP SP3 32-bit version and Java version 1.6.0_24. The computer running the Coordinator and the Code Server was an AMD Opteron 2216 HE 2.4GHz (dual core) with 2GB of RAM and SUSE Linux with Kernel version 2.6.31.12. The Clients were run on a normal workstation with a configuration similar to the Workers. All computers were connected through a switched 1 Gbps Ethernet. The network topology can be seen in Figure 33.

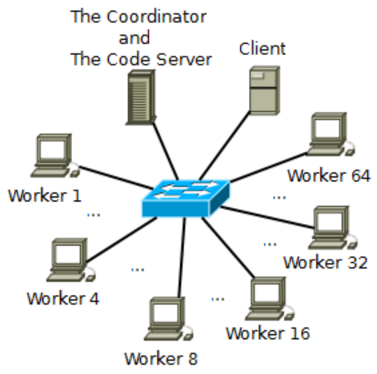


Figure 33: Network topology used for the experiments

5.2.2 Used metrics

Each experiment was repeated five times and the results below are averages of the results collected. Each job was run with different numbers of cores: 8, 16, 32, 64 and 128. All the experiments were done under similar conditions and according to the same procedure. In the presentation of the results, it is use the following terms:

- *Number of cores*: total number of CPU cores used to perform an experiment. For this study, we used 8, 16, 32, 64 and 128 cores.
- *Average runtime*: average runtime observed in five experiments with a particular job and with a certain number of cores.
- *Speed up*: speed up gained when doubling the number of cores. The speed up is calculated by dividing the average runtime for n cores by the average runtime of $2n$ cores.

5.2.3 Pi

Pi Client experiment calculates 10,000 decimals of π, divided into 200 tasks. Each task is the computation 50 decimals of π. The average runtime of the Pi job with different numbers of cores is plotted in Figure 34. The timings and the speed ups are listed in Table 2. The results show that computation of decimals of scales well with an average speed up of x1.9 each time the number of cores is doubled. To calculate 10,000 decimal places of π with 8 cores took an average of 242.6 seconds, while it took only 19.4 seconds with 128 cores. The total network data transferred from and to the server was approximately 4 MB in both directions.

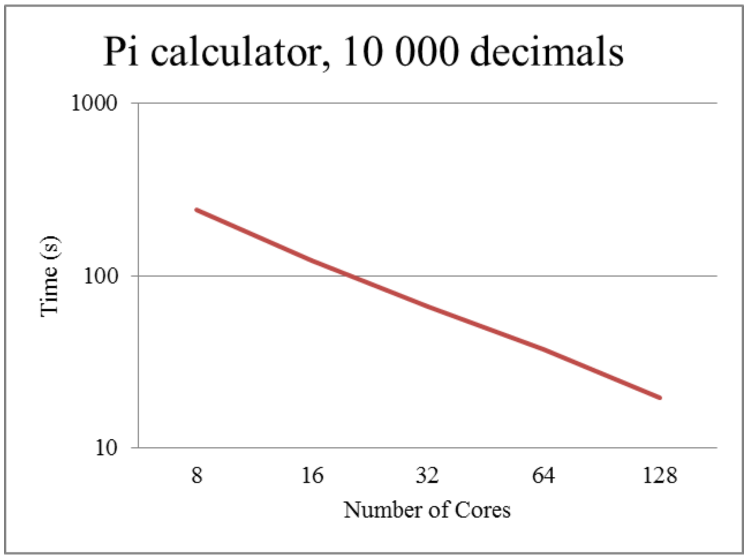


Figure 34: Average runtime in 5 runs for each set of cores of the Pi Client

Number of cores	8	16	32	64	128
Average time(s)	287.6	149.3	76.1	50.6	28.6
Speed-up		x2.0	x1.9	x1.8	x1.9

Table 2: Average runtime in 5 runs of each set of cores for the Pi Client

5.2.4 Mandelbrot

For this experiment a 4096 interations has been used and a total of 1600 tasks have been created. The average runtime for the Mandelbrot job are plotted in Figure 35. The timings and the speed up for the Mandelbrot job are shown in Table 3. The Mandelbrot job does not scale as well as the Pi job described in Section 5.2.3. It took 128.0 seconds to calculate the 1000 partial images on 8 cores and 19.0 seconds on 128 cores. Speed up decreases as the number of cores is increased. We observed an average speed up of x1.9 going from 8 cores to 16 cores. However, going from 64 cores to 128 cores, we only observed a speed up of x1.3. The reason why the Mandelbrot job does not scale well when the number of cores becomes larger than 32 is the heterogeneous nature of the tasks: in the Mandelbrot job, the amount of computation necessary to generate each the partial image depends heavily on the local region of the Mandelbrot set that an image represents. The difference in terms of the amount of computation necessary to compute two partial images from different local regions is up to three orders of magnitude. When Conillon has access to more than 32 cores, it can happen that a large fraction of the Workers become idle towards the end of the job while one or a few of the Workers are still calculating one of the more complex tasks. As can be seen in Figure 35, this effect starts to have a significant impact on the average runtime around 64 cores, where the plot starts to have an easier slope. This also means that, for the Mandelbrot job, adding more cores would most likely yield only a minimal gain in terms of performance, because the limiting factor is the few tasks that require significantly more computation than most the others. A possible solution for this problem would be to divide the Mandelbrot job into a larger number of smaller tasks – tasks could, for example, be to compute partial images of 100x100 pixels instead of 200x200 pixels. This would make the distributed computation more fine grained and in turn allow Conillon to distribute the job more evenly in between the Workers.

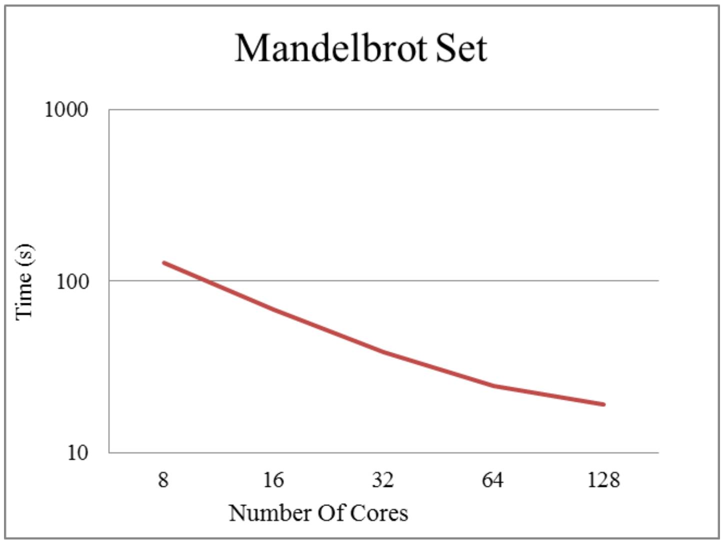


Figure 35: Average runtime in 5 runs of each set of cores for the Mandelbrot Set Client

Number of cores	8	16	32	64	128
Average time(s)	128.0	68.1	38.3	24.4	19.0
Speed-up		x1.9	x1.8	x1.6	x1.3

Table 3: Average runtime in 5 runs of each set of cores for the Mandelbrot Client

5.2.5 Image Processing

For the Image Processing job, the average runtime are plotted in Figure 36. The timings and the speed up are listed in Table 4. As it can be seen from the results, the Image Processing job scales quite well with an average speed up of x1.9 each time the number of cores is double. It took 1138.0 seconds to process the 1000 images with 8 cores and only 93.2 seconds with 128 cores. We registered the network data transferred from and to the server for this job (around 125 MB in both directions).The comparatively large amount of network transfers is because image are sent two and from the Client and to and from the Workers.

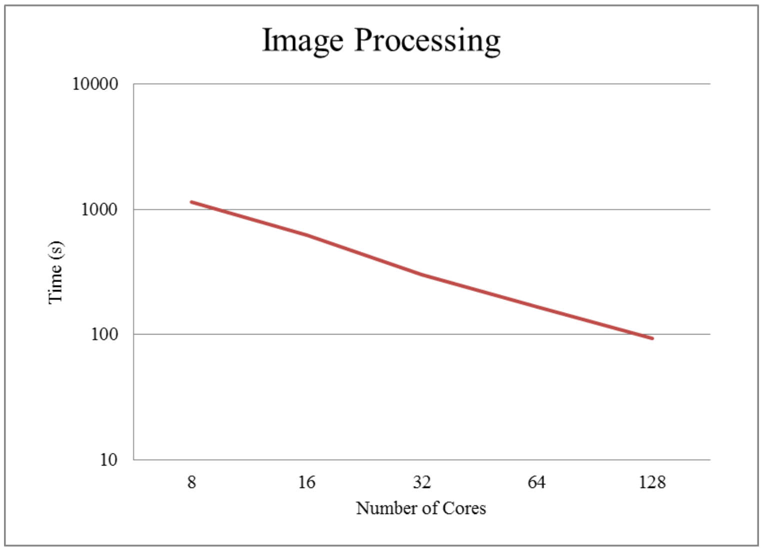


Figure 36: Average runtime in 5 runs of each set of cores for the Image Processing Client

Number of cores	8	16	32	64	128
Average time(s)	1138.0	626.4	310.0	166.9	93.2
Speed-up		x1.8	x2.0	x2.0	x1.9

Table 4: Average runtime in 5 runs of each set of cores for the Image Processing Client

5.2.6 P72

For this experiment is was used an total of 900,000 numbers divided in 1000 intervals. Each interval is a task. The results for the P72 average runtime are plotted in Figure 37. The values for the runtime and the speed up are listed in Table 5. For this job, we observed an average speed up with of x1.8 each time the number of cores was doubled. The 1000 tasks were calculated in 235.7 seconds by 8 cores and in 22.7 seconds by 128 cores.

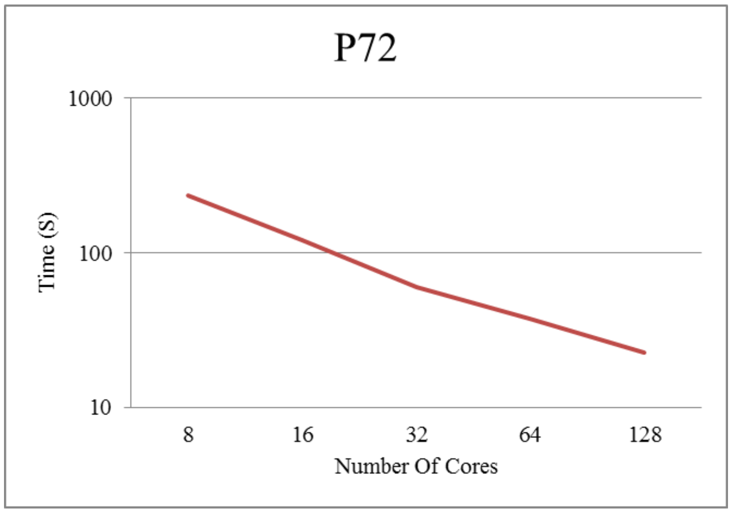


Figure 37: Average runtime in 5 runs with each set of cores for the P72 Client

Number of cores	8	16	32	64	128
Average time(s)	235.7	119.8	59.9	37.5	22.7
Speed-up		x2.0	x2.0	x1.6	x1.7

Table 5: Average runtime in 5 runs with each set of cores for the P72 Client

5.2.7 Four Clients running concurrently

In this set of experiments, we let all the four Clients, with the same configurations, submit their job at the same time. This stresses the Coordinator that has to receive, schedule and distribute several tasks at the same time. The coordinator receives four jobs with a total of 4100 tasks from the Clients in a short period of time and immediately starts distributing them. The average runtime of this experiment with different numbers of cores is plotted in Figure 38. As can be seen at Table 6, the speed up is relatively high (x1.8) each time the number of cores is doubled. When multiple Clients are running at the same time, Conillon can benefit from having more tasks to distribute: even if some tasks take much longer than others, the Workers do not become idle when the Mandelbrot job is nearing completion – instead, the Workers are given tasks from the other jobs. The average runtime was 1638.5 seconds in the experiments where we used 8 cores and 150.0 seconds in the experiments where we used 128 cores.

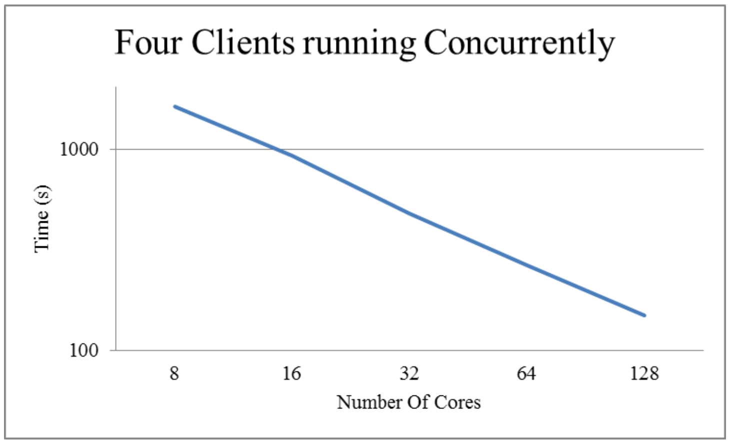


Figure 38: Average runtime in 5 runs of the four Clients running concurrently

Number of cores	8	16	32	64	128
Average time(s)	1638.5	933.9	480.0	263.7	150.0
Speed-up		x1.8	x1.9	x1.8	x1.8

Table 6: Average runtime in 5 runs of each set of cores for the four Clients running concurrently

5.2.8 Results summary

The results presented in this section show that Conillon scales well when computing a set of different jobs with distinct characteristics. When the number of tasks is large, Conillon can distribute the computation among the Workers evenly and thereby minimize the time that Workers are idle. In Figure 39, we have plotted the average runtime when all the jobs are running concurrently with the sum of the average runtime of all the jobs running independently. As it can be seen in Figure 39, when the Clients all run at the same time, the average runtime is lower than the sum of the average runtime when the Clients run independently.

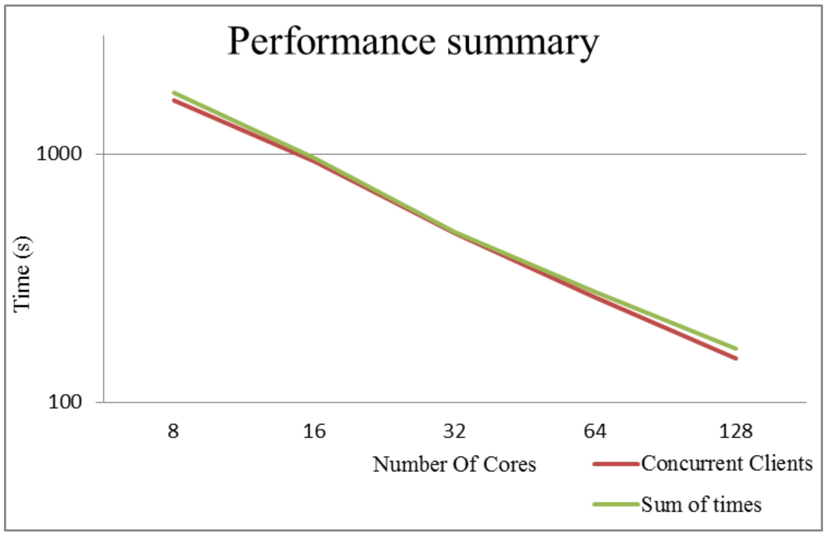


Figure 39: Performance summary – sums of average times when run individually vs. running all Clients concurrently

5.3 Summary

In this chapter is described the developed applications used to test Conillon. The goal of this chapter is to obtain an actual performance evaluation obtained through the various experiments. Our experiments involved four different types of jobs with distinct characteristics. The main differences between the jobs were the size of the individual tasks, the results, and the computational time to execute each task. In the Mandelbrot experiments, the speed up decreased when the number of cores was increased beyond 32. This is due to the fact that the tasks in this job vary greatly in terms of computational complexity. As a result, a few demanding tasks become the bottleneck for the job. This issue can be overcome by dividing each job into a larger number of smaller tasks at the expense of more communication overhead. This situation can cause bottlenecks at the servers or in the network equipment. This Chapter proved that our programming model is flexible, easy to use, and with enormous capabilities.

6 Conclusion

The main objective of this work is to develop an distributed computing platform named Conillon. The expected goals are: *Research and develop a distributed computing platform. Scalable - dynamic addition of nodes, Maximize the use of the CPU, Fault-tolerant engine, Ability to cope with multiple problems simultaneously, Easy to use and manage, Support for different programming languages, Maximize performance and efficiency, User accessibility: Multi-platform Applet - ability to use the web browser to explore the CPU cycles, Screen-saver application - to use in the different Operative Systems;*

We were able to meet all our objectives. The simple and efficient architecture proved to be performant, scalable and easy to use. The servers start with little user effort and the Conillon programming model is easy to learn, straightforward, and gives the ability to develop parallel applications without worrying about distribution and/or synchronization issues. Conillon requires little maintenance, the system is self-sufficient in the failures and in management. The Worker application is designed to fully explore CPU resources with no need for user administration. Both servers, Coordinator and Code server, and the Worker can be used in any JVM enabled Operating System like Linux, Solaris, Windows and MacOS X. The Worker can even be run in a Web Browser for sporadic exploration of resources, or can be installed the stand-alone application with integrated screen saver.

The presented results show that all the experiments scale relatively well and we observed an average speed up of up to x1.9 each time the number of cores was doubled. Task programming requires focus, tasks must be similar and with more fine grained complexity to adapt well to the heterogeneous conditions of a grid. See the case of our Mandelbrot experiment at Section 5.2.4. Meanwhile, having a larger number of tasks can cause a bottleneck at the network equipment or at the servers.

To summarize, the objectives were fulfilled and Conillon is able to provide an easy to use programming model, simple to use, install and the management is self-sufficient with little user administration. Is the author's opinion that the project is suitable to be installed and exploited in production environments.

The source code is available under BSD[7] license and can be downloaded at:
http://code.google.com/p/distributedcomputing/.

6.1 Future Work

In our ongoing work, we are including some of the new technologies that are becoming available for Java to take advantage of modern GPUs such as OpenCL[34] while still keeping Conillon lightweight and straightforward for small and medium organizations to use.

Other area of major improvement is an intelligent scheduler [22]. A sophisticate scheduler will allow the platform to distribute tasks more efficiently to the available resources. E.g., The platform could schedule the most complicated tasks for the more powerful resources in a group of heterogeneous workstations. This way is possible to do a better timing optimization, getting faster the results.

[7]BSD see http://www.opensource.org/licenses/bsd-license.php for more information about BSD license

7 Publications

- Silva, H., Christensen, A., and Oliveira, S. Building and designing a distributed computing platform. In Proceedings of the Workshop on Open Source and Design of Communication (New York, NY, USA, 2010), OSDOC '10, ACM, pp. 5558.
- Silva, H., Oliveira, S., and Christensen, A. Conillon: A lightweight distributed computing platform for desktop grids. 6th Conferência Ibérica de Sistemas e Tecnologias de Informação. Chaves, Portugal
- Silva, H., Christensen, A., and Oliveira, S. Performance study of conillon - a platform for distributed computing. In Proceedings of the Workshop on Open Source and Design of Communication (New York, NY, USA, 2011), OSDOC '11, ACM.

References

[1] High performance parametric modeling with nimrod: Killer application for the global grid? In *Proceedings of the 14th International Symposium on Parallel and Distributed Processing* (Washington, DC, USA, 2000), IEEE Computer Society, pp. 520–.

[2] Alexandrov, A. D., Ibel, M., Schauser, K. E., and Scheiman, C. J. Superweb: Research issues in java-based global computing, 1996.

[3] Anderson, D. P. Boinc: A system for public-resource computing and storage. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing* (Washington, DC, USA, 2004), GRID '04, IEEE Computer Society, pp. 4–10.

[4] Anderson, D. P., Christensen, C., and Allen, B. Designing a runtime system for volunteer computing. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing* (New York, NY, USA, 2006), SC '06, ACM.

[5] Anderson, D. P., Cobb, J., Korpela, E., Lebofsky, M., and Werthimer, D. Seti@home: an experiment in public-resource computing. *Commun. ACM 45* (November 2002), 56–61.

[6] Bailey, D., Borwein, P., and Plouffe, S. On the rapid computation of various polylogarithmic constants. *Mathematics of Computation 66* (1996), 903–913.

[7] Buyya, R., and Abramson, D. The virtual laboratory: a toolset to enable distributed molecular modelling for drug design on the world-wide grid, 2003.

[8] Cappello, F., Djilali, S., Fedak, G., Herault, T., Magniette, F., Néri, V., and Lodygensky, O. Computing on large-scale distributed systems: Xtrem web architecture, programming models, security, tests and convergence with grid. *Future Gener. Comput. Syst. 21* (March 2005), 417–437.

[9] Cappello, P., Christiansen, B., Ionescu, M. F., Neary, M. O., Schauser, K. E., and Wu, D. Javelin: Internet-based parallel computing using java, 1997.

[10] Diomidis, K. R., Raptis, K., Spinellis, D., Katsikas, S., and Gr-Karlovassi. Java as distributed object glue, 2000.

[11] Ferreira, L., Berstis, V., Armstrong, J., Kendzierski, M., Neukoetter, A., MasanobuTakagi, Bing, R., Amir, A., Murakawa, R., Hernandez, O., Magowan, J., and Bieberstein, N. *Introduction to grid computing with globus*, first ed. IBM Corp., Riverton, NJ, USA, 2003.

[12] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T. Rfc 2616, hypertext transfer protocol – http/1.1, 1999.

[13] Foster, I., and Kesselman, C., Eds. *The grid: blueprint for a new computing infrastructure.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[14] Goldchleger, A., Kon, F., Goldman, A., Finger, M., and Bezerra, G. C. Integrade object-oriented grid middleware leveraging the idle computing power of desktop machines: Research articles. *Concurr. Comput. : Pract. Exper. 16* (April 2004), 449–459.

[15] Heath, T. L., and Euclid. *The Thirteen Books of Euclid's Elements, Books 10–13.* Dover Publications, June 1956.

[16] Juneau, J., Baker, J., Wierzbicki, F., Soto, L., and Ng, V. *The Definitive Guide to Jython: Python for the Java Platform*, 1st ed. Apress, Berkely, CA, USA, 2010.

[17] Lewandowski, S. M. Frameworks for component-based client/server computing. *ACM Comput. Surv. 30* (March 1998), 3–27.

[18] Mandelbrot, B. B. *The fractal geometry of nature.* W. H. Freeman, New York, 1983.

[19] Marr, D. T., Binns, F., Hill, D. L., Hinton, G., Koufaty, D. A., Miller, J. A., and Upton, M. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal 6*, 1 (2002), 1–12.

[20] Minoli, D. *A Networking Approach to Grid Computing.* Wiley-Interscience, 2004.

[21] Neary, M. O., Phipps, A., Richman, S., and Cappello, P. Javelin 2.0: Java-based parallel computing on the internet. In *Internet, Proceedings of European Parallel Computing Conference (Euro-Par 2000* (2000), Springer, pp. 1231–1238.

[22] Ni, L., Zhang, J., Yan, C., and Jiang, C. A heuristic algorithm for task scheduling based on mean load. In *Proceedings of the First International Conference on Semantics, Knowledge and Grid* (Washington, DC, USA, 2005), SKG '05, IEEE Computer Society, pp. 5–.

[23] Odersky, M., and al. An overview of the scala programming language. Tech. Rep. IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[24] Oliveira, L., Lopes, L., Silva, F., Peer, P. P. T., Silva, O., and Alegre, R. D. C. P&sup3;: Parallel peer to peer - an internet parallel programming environment.

[25] Postel, J. Transmission Control Protocol. RFC 793 (Standard), Sept. 1981. Updated by RFCs 1122, 3168.

[26] Ripeanu, M., Foster, I., and Iamnitchi, A. Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system design. *IEEE Internet Computing Journal 6* (2002), 2002.

[27] Sahni, S. Scheduling master-slave multiprocessor systems. *IEEE Trans. Comput. 45* (October 1996), 1195–1199.

[28] Sarmenta, L. F. G., and Hirano, S. Bayanihan: Building and studying web-based volunteer computing systems using java. 675–686.

[29] Satoshi, H. Horb: Distributed execution of java programs. In *Proceedings of the International Conference on Worldwide Computing and Its Applications* (London, UK, 1997), WWCA '97, Springer-Verlag, pp. 29–42.

[30] Segal, B., Robertson, L., Gagliardi, F., and Carminati, F. Grid computing: the european data grid project.

[31] Silva, H., Christensen, A., and Oliveira, S. Building and designing a distributed computing platform. In *Proceedings of the Workshop on Open Source and Design of Communication* (New York, NY, USA, 2010), OSDOC '10, ACM, pp. 55–58.

[32] Silva, H., Christensen, A., and Oliveira, S. Performance study of conillon - a platform for distributed computing. In *Proceedings of the Workshop on Open Source and Design of Communication* (New York, NY, USA, 2011), OSDOC '11, ACM.

[33] Silva, H., Oliveira, S., and Christensen, A. Conillon: A lightweight distributed computing platform for desktop grids. 6th Conferência Ibérica de Sistemas e Tecnologias de Informação.

[34] Stone, J. E., Gohara, D., and Shi, G. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in Science and Engineering 12* (2010), 66–73.

[35] Tanenbaum, A. S., and Steen, M. v. *Distributed Systems: Principles and Paradigms (2nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[36] Taurus, D. G. H. A taxonomy of actual utilization of real unix and windows servers. Tech. rep., IBM, 2003.

[37] Thain, D., Tannenbaum, T., and Livny, M. Distributed computing in practice: the condor experience: Research articles. *Concurr. Comput. : Pract. Exper. 17* (February 2005), 323–356.

[38] Tsirtsis, G., and Srisuresh, P. Network address translation - protocol translation (nat-pt), 2000.

[39] Ulrik, C., Jakob, S., and Pedersen, G. Developing distributed computing solutions combining grid computing and public computing m.sc. thesis, 2005.

[40] Wollrath, A., Waldo, J., and Riggs, R. Java-centric distributed computing. *IEEE Micro 17* (May 1997), 44–53.

8 Source code

8.1 Pi

8.1.1 PiClient class

```java
package mypi;
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.math.BigDecimal;
import comm.ClientPriority;

import client.Client;
public class PiClient extends Client {
        //max scale for the Pi number used for this experiment
        private static final int SCALE = 10000;
        //work size for each task
        private static final int WORK_SIZE = 50;

        public PiClient(ClientPriority priority, int problemNumber, int version,String
            masterAddress, int masterPort, String codeServerAddress,int codeServerPort) {
                super(priority, problemNumber, version, masterAddress, masterPort,
                    codeServerAddress, codeServerPort);
        }

        public void execute() {
                long time = System.currentTimeMillis();
                BigDecimal result = new BigDecimal("0");
                //Setting the scale for the Pi number, in this case 10 000 decimals
                result = result.setScale(SCALE);
                int numIterations = SCALE / WORK_SIZE;
                for (int i = 0; i < numIterations; i++) {
                        //Creating the tasks, each one with fixed work size = 50, with 10
                            000 decimals scale
                        PiTask piCalc = new PiTask(i, i * WORK_SIZE, (i + 1) * WORK_SIZE,
                            SCALE);
                        //inject task to the Coordinator
                        commit(piCalc);
                }

                for (int i = 0; i < numIterations; i++) {
                        //Getting ready results and sum to obtain the final Pi number
                        result = result.add(((MyResult) (getNextResult())).getValue());
                }
                try{
                          // Create file
                          FileWriter fstream = new FileWriter("out.txt");
                          BufferedWriter out = new BufferedWriter(fstream);
                          out.write(result.toString());
                          //Close the output stream
                          out.close();
                          }catch (Exception e){//Catch exception if any
                          System.err.println("Error:␣" + e.getMessage());
                          }
        }
        public static void main(String[] args) {
                //creating the client with very high priority, problem number and version 6,
                     and the conillon's address
                new PiClient(ClientPriority.VERY_HIGH , 6, 6, "127.0.0.1", 0, "127.0.0.1",
                    0).execute();
        }

}
```

Listing 1: PiClient class

8.1.2 PiTask class

```
package mypi;
import java.math.BigDecimal;
import java.math.BigInteger;
import result.Result;
import tasks.Task;

public class PiTask extends Task {
        //setting the necessary attributes for the the calculus of Pi - the values are
            constants and can be found on Bailey-Borwein-Plouffe article
        private static final int ROUND_MODE = BigDecimal.ROUND_HALF_EVEN;
        private BigDecimal ZERO = new BigDecimal("0");
        private BigDecimal ONE = new BigDecimal("1");
        private BigDecimal NEGATIVE_ONE = new BigDecimal("-1");
        private BigDecimal TWO  = new BigDecimal("2");
        private BigDecimal NEGATIVE_TWO = new BigDecimal("-2");
        private BigDecimal FOUR = new BigDecimal("4");
        private BigDecimal FIVE = new BigDecimal("5");
        private BigDecimal SIX = new BigDecimal("6");
        private BigDecimal EIGHT = new BigDecimal("8");
        private BigDecimal NEGATIVE_EIGHT = new BigDecimal("-8");
        private BigInteger SIXTEEN = new BigInteger("16");
        private MyResult result;
        private int start;
        private int end;
        int scale;

        public PiTask(int id, int start, int end, int scale) {
                super(id);
                this.start = start;
                this.end = end;
                this.scale = scale;

        }
        public PiTask(){
                super(0);
        }
        // Bailey-Borwein-Plouffe Pi Algorithm
        private BigDecimal f(int k) {
                BigDecimal K = new BigDecimal(k);
                BigDecimal EIGHT_K = EIGHT.multiply(K);
                BigDecimal FIRST = ONE.divide(new BigDecimal(SIXTEEN.pow(k)), ROUND_MODE);
                BigDecimal SECOND = FOUR.divide(EIGHT_K.add(ONE), ROUND_MODE);
                BigDecimal THIRD = NEGATIVE_TWO.divide(EIGHT_K.add(FOUR), ROUND_MODE);
                BigDecimal FOURTH = NEGATIVE_ONE.divide(EIGHT_K.add(FIVE), ROUND_MODE);
                BigDecimal FIFTH = NEGATIVE_ONE.divide(EIGHT_K.add(SIX), ROUND_MODE);

                return FIRST.multiply(SECOND.add(THIRD.add(FOURTH.add(FIFTH))));
        }
        //setting scale for the calculated number, in this case 10 000
        public void setScale(Integer scale) {
                ZERO = ZERO.setScale(scale);
                ONE = ONE.setScale(scale);
                NEGATIVE_ONE = NEGATIVE_ONE.setScale(scale);
                TWO = TWO.setScale(scale);
                NEGATIVE_TWO = NEGATIVE_TWO.setScale(scale);
                FOUR = FOUR.setScale(scale);
                FIVE = FIVE.setScale(scale);
                SIX = SIX.setScale(scale);
                EIGHT = EIGHT.setScale(scale);
                NEGATIVE_EIGHT = NEGATIVE_EIGHT.setScale(scale);
        }
        //starting the calculus
        private BigDecimal start_cal(){
                BigDecimal bd = ZERO;
                BigDecimal total = ZERO;
                for (int i = start ;i<end;i++){
                        bd = bd.add(f(i));
                }
                return bd;
        }

        //Run method, method that runs on workers
        @Override
        public void run() {
                //setting scale
                setScale(new Integer(scale));
                //calculating the result for the work size (50) with 10000 scale
                result = new MyResult( start_cal());

        }

        //Method that returns the result to clients
        @Override
        public Result getResult() {
                return result;
        }
        @Override
        public String toString() {
                return "Simple␣PI␣Task␣:)";
        }


}
```

Listing 2: PiTask class

8.1.3 PiResult class

```
package mypi;
import java.math.BigDecimal;
import result.Result;
public class MyResult extends Result {
        //attribute that will store the calculated result, in this case the Pi number
        private BigDecimal value;

        public MyResult(BigDecimal bigDecimal) {
                super();
                this.value = bigDecimal;
        }
        //returning the value to client
        public BigDecimal getValue() {
                return value;
        }
```

```java

	public void setValue(BigDecimal value) {
		this.value = value;
	}

	@Override
	public String toString() {
		return "MyResult [value=" + value + "]";
	}


}
```

Listing 3: PiResult class

8.2 Image Processing

8.2.1 ImageProcessingClient class

```java
package myImage;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;

import client.Client;

import comm.ClientPriority;

public class ImageProcessingClient {

	Client client;
	private int numberOfTasks;
	private String[] files;

	public ImageProcessingClient() {
		//Creating the client
		client = new Client(ClientPriority.VERY_HIGH, 2, 2, "127.0.0.1", 0,"
		    127.0.0.1", 0);
	}

	private void init() {
		//Getting the images
		File sourceimage = new File("1000images");
		files = sourceimage.list();
		numberOfTasks = files.length;
		long time = System.currentTimeMillis();
		//Thread to submmit tasks to the Coordinator
		new Thread(new Runnable() {
			@Override
			public void run() {
				for (int i = 0; i < numberOfTasks; i++) {
					try {
						byte[] image = getBytesFromFile(new File("
						    1000images"+ File.separator + files[i]))
						    ;
						//creating task with the image and
						    specifying the matrix size = 8
```

```
                        ImageProcessingTask task = new ImageProcessingTask(i,8,image);
                        //injecting the created task in the Coordinator
                        client.commit(task);
                    } catch (FileNotFoundException e) {
                        e.printStackTrace();
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                }
            }

        }).start();
        //Getting the results
        for (int i = 0; i < numberOfTasks; i++) {
            ImageProcessingResult result = (ImageProcessingResult) client.getNextResult();
            try {
                //Saving the image received
                FileOutputStream out = new FileOutputStream("ImageResults" + File.separator + "output"+ result.getId()+".jpg");
                out.write(result.getImage());
                out.close();
            } catch (FileNotFoundException e) {
                e.printStackTrace();
            } catch (IOException e) {
                e.printStackTrace();
            }
            System.out.println(result.getId() + ":" +(System.currentTimeMillis()-time));
        }
        System.out.println("Total time:" + (System.currentTimeMillis()-time));
        System.exit(0);
    }
    //Method that return and array of bytes from a given File
    public byte[] getBytesFromFile(File file) throws IOException {
        InputStream is = new FileInputStream(file);
        long length = file.length();
        if (length > Integer.MAX_VALUE) {

        }
        byte[] bytes = new byte[(int) length];

        int offset = 0;
        int numRead = 0;
        while (offset < bytes.length
                && (numRead = is.read(bytes, offset, bytes.length - offset)) >= 0) {
            offset += numRead;
        }
        if (offset < bytes.length) {
            throw new IOException("Could not completely read file " + file.getName());
        }
        is.close();
        return bytes;
    }

    public static void main(String[] args) {
        new ImageProcessingClient().init();
    }
```

```
}
```

Listing 4: ImageProcessingClient class

8.2.2 ImageProcessingTask class

```
package myImage;


import java.awt.Graphics2D;
import java.awt.Image;
import java.awt.image.BufferedImage;
import java.awt.image.BufferedImageOp;
import java.awt.image.ConvolveOp;
import java.awt.image.Kernel;
import java.io.ByteArrayOutputStream;
import java.io.IOException;

import javax.imageio.ImageIO;
import javax.swing.ImageIcon;

import result.Result;
import tasks.Task;

public class ImageProcessingTask extends Task {

        private int DIM;
        private byte[] image;

        public ImageProcessingTask(int id, int dim, byte[] image) {
                super(id);
                this.image = image;
                this.DIM = dim;
        }
        //Run method, this method is going to be executed at the workers
        @Override
        public void run() {
                try {
                        ImageIcon im = new ImageIcon(image);
                        Image source = im.getImage();
                        int w = source.getWidth(null);
                        int h = source.getHeight(null);
                        //creating the buffer to the image
                        BufferedImage sourceImage = new BufferedImage(w, h,
                            BufferedImage.TYPE_INT_RGB);
                        //Necessary class to render the image
                        Graphics2D g2d = (Graphics2D) sourceImage.getGraphics();
                        g2d.drawImage(source, 0, 0, null);
                        g2d.dispose();
                        //Creating the matrix to apply Convolution
                        double[][] matrix = new double[DIM][DIM];
                        for (int i = 0; i < DIM; i++)
                                for (int j = 0; j < DIM; j++)
                                        matrix[i][j] = 1.0f / (DIM * DIM);
                        //Applying convolution with the given matrix size to the image
                            buffer
                        BufferedImage blurredImage = Convolution.convolution2D(sourceImage,
                            sourceImage.getWidth(), sourceImage.getHeight(), matrix,DIM, DIM
                            );
                        ByteArrayOutputStream stream = new ByteArrayOutputStream(200000);
                ImageIO.write(blurredImage, "jpg", stream);
                stream.flush();
                stream.close();
                //saving the image to be returned by the getResult() method
                image = stream.toByteArray();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }

        @Override
        public Result getResult() {
            return new ImageProcessingResult(id, image);
        }
}
```

Listing 5: ImageProcessingTask class

8.2.3 ImageProcessingResult class

```
package myImage;
import result.Result;


public class ImageProcessingResult extends Result{
        private int id;
        //Result of the calculated task, in this case an array of bytes
        private byte[] image;
        public ImageProcessingResult(int id, byte[] image) {
            super();
            this.id = id;
            this.image = image;
        }
        public int getId() {
            return id;
        }
        public byte[] getImage() {
            return image;
        }
}
```

Listing 6: ImageProcessingResult class

8.3 Mandelbrot

8.3.1 MandelbrotClient class

```
package mandelbrot;

import java.awt.Color;
import java.awt.Graphics2D;
import java.awt.Image;
import java.awt.image.BufferedImage;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

import javax.imageio.ImageIO;
import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JLabel;

import client.Client;

import comm.ClientPriority;

public class MandelbrotClient {
        private Client client;
        private int maxCount = 8192;
        private boolean smooth = false;
        private boolean antialias = false;
        private boolean drag = false;
        private boolean toDrag = false;
        private boolean rect = true, oldRect = true;
        private Color[][] colors;
        private int pal = 0;
        private double viewX = 0.0;
        private double viewY = 0.0;
        private double zoom = 1.0;
        private int width  = 8000;
        private int height = 8000;
        private int boxHeight = 200;
        private int boxWidth  = 200;
        private JFrame frame;
        private BufferedImage image;
        private static final int[][][] colpal = {
                { {0, 10, 20}, {50, 100, 240}, {20, 3, 26}, {230, 60, 20},
                        {25, 10, 9}, {230, 170, 0}, {20, 40, 10}, {0, 100, 0},
                        {5, 10, 10}, {210, 70, 30}, {90, 0, 50}, {180, 90, 120},
                        {0, 20, 40}, {30, 70, 200} },
                        { {70, 0, 20}, {100, 0, 100}, {255, 0, 0}, {255, 200, 0} },
                        { {40, 70, 10}, {40, 170, 10}, {100, 255, 70}, {255, 255, 255} },
                        { {0, 0, 0}, {0, 0, 255}, {0, 255, 255}, {255, 255, 255}, {0, 128,
                            255} },
                        { {0, 0, 0}, {255, 255, 255}, {128, 128, 128} },
        };

        private void init() {
                client = new Client(ClientPriority.LOW, 20, 2, "127.0.0.1", 0,"127.0.0.1",
                    0);
                long time = System.currentTimeMillis();
                System.out.println("Started at:" + time);
                // initialize color palettes
                colors = new Color[colpal.length][];
                for (int p = 0; p < colpal.length; p++) {
                        colors[p] = new Color[colpal[p].length * 12];
                        for (int i = 0; i < colpal[p].length; i++) {
                                int[] c1 = colpal[p][i];
                                int[] c2 = colpal[p][(i + 1) % colpal[p].length];
                                for (int j = 0; j < 12; j++)
                                        colors[p][i * 12 + j] = new Color(
                                                        (c1[0] * (11 - j) + c2[0] * j) / 11,
                                                        (c1[1] * (11 - j) + c2[1] * j) / 11,
                                                        (c1[2] * (11 - j) + c2[2] * j) / 11)
                                                            ;
                        }
                }
                //Creating a buffer to the final image
```

```java
        image = new BufferedImage((int) width, (int) height, BufferedImage.TYPE_INT_RGB);
        new Thread(new Runnable() {
            @Override
            public void run() {
                int  currentBox = 0;
                for (int by = 0; by < height / boxHeight; by++) {
                    for (int bx = 0; bx < width / boxWidth; bx++) {
                        //committing the created task with the 200 per 200 size
                        client.commit(new MandelbrotTask(currentBox++,by,bx,boxWidth,boxHeight,zoom,width,height,viewX,viewY,colors,maxCount,System.currentTimeMillis()));
                    }
                }
            }
        }).start();

        Graphics2D g2d = (Graphics2D)image.getGraphics();
        int numberOfBoxes = height / boxHeight * width / boxWidth;
        //Assembling the final image with the portions received by each worker tasks
        for (int i = 0; i < numberOfBoxes ; i++) {
            MandelbrotResult result = (MandelbrotResult) client.getNextResult();
            ImageIcon im =  new ImageIcon(result.getImage());
            Image source = im.getImage();
            g2d.drawImage(source, result.getX()*boxWidth, result.getY()*boxHeight, null);
        }
        try {
            //saving the Mandelbrot image
            FileOutputStream out = new FileOutputStream("result.jpg");
            ImageIO.write(image,"jpg", out);
            out.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
        System.exit(0);
    }

    public static void main(String[] args) {
        new MandelbrotClient().init();
    }
}
```

Listing 7: MandelbrotClient class

8.3.2 MandelbrotTask class

```java
package mandelbrot;

import java.awt.Color;
import java.awt.image.BufferedImage;
import java.io.ByteArrayOutputStream;
import java.io.IOException;

import javax.imageio.ImageIO;

import result.Result;
import tasks.Task;

public class MandelbrotTask extends Task {

        private int by;
        private int bx;
        private int boxWidth;
        private int boxHeight;
        private double zoom;
        private int width;
        private int height;
        private double viewX;
        private double viewY;
        private BufferedImage image;
        private int pal;
        private Color[][] colors;
        private boolean smooth = false;
        private int maxCount;
        private byte[] result;
        private long initTime;
        private long startTime;

        public MandelbrotTask(int id, int by, int bx, int boxWidth, int boxHeight,double
            zoom, int width, int height, double viewX, double viewY,Color[][] colors, int
            maxCount, long time) {
                super(id);
                this.by = by;
                this.bx = bx;
                this.boxWidth = boxWidth;
                this.boxHeight = boxHeight;
                this.zoom = zoom;
                this.width = width;
                this.height = height;
                this.viewX = viewX;
                this.viewY = viewY;
                this.colors = colors;
                this.maxCount = maxCount;
                this.initTime = time;
        }
        //Method that is executed at the workers
        @Override
        public void run() {
                try {
                        startTime=System.currentTimeMillis();
                        //creating the image - t his image is a small portion of the final
                            image
                        image = new BufferedImage(boxWidth, boxHeight, BufferedImage.
                            TYPE_INT_RGB);
                        int offY = by * boxHeight;
                        int offX = bx * boxWidth;
                        //calculates the point and applys the color
                        for (int y = offY; y < (by + 1) * boxHeight; y++) {
                                for (int x = offX; x < (bx + 1) * boxWidth; x++) {
                                        double r = zoom / (double) Math.min(width, height);
                                        double dx = 2.5 * (x * r + viewX) - 2;
                                        double dy = 1.25 - 2.5 * (y * r + viewY);
                                        Color color = color(dx, dy);
                                        image.setRGB(x - offX, y - offY, color.getRGB());
                                }
                        }
                        ByteArrayOutputStream stream = new ByteArrayOutputStream(200000);
```

```
                    ImageIO.write(image, "jpg", stream);
                    stream.flush();
                    stream.close();
                    //saves the image
                    result = stream.toByteArray();
            } catch (Exception e) {
                    e.printStackTrace();
            }
    }
    //return the computed image and the coodinates of the portion
    @Override
    public Result getResult() {
            return new MandelbrotResult(super.id,bx, by, result,startTime,initTime);
    }

    // Computes a color for a given point
    private Color color(double x, double y) {
            int count = mandel(0.0, 0.0, x, y);
            int palSize = colors[pal].length;
            Color color = colors[pal][count / 256 % palSize];
            if (smooth) {
                    Color color2 = colors[pal][(count / 256 + palSize - 1) % palSize];
                    int k1 = count % 256;
                    int k2 = 255 - k1;
                    int red = (k1 * color.getRed() + k2 * color2.getRed()) / 255;
                    int green = (k1 * color.getGreen() + k2 * color2.getGreen()) / 255;
                    int blue = (k1 * color.getBlue() + k2 * color2.getBlue()) / 255;
                    color = new Color(red, green, blue);
            }
            return color;
    }

    // Computes a value for a given complex number
    private int mandel(double zRe, double zIm, double pRe, double pIm) {
            double zRe2 = zRe * zRe;
            double zIm2 = zIm * zIm;
            double zM2 = 0.0;
            int count = 0;
            while (zRe2 + zIm2 < 4.0 && count < maxCount) {
                    zM2 = zRe2 + zIm2;
                    zIm = 2.0 * zRe * zIm + pIm;
                    zRe = zRe2 - zIm2 + pRe;
                    zRe2 = zRe * zRe;
                    zIm2 = zIm * zIm;
                    count++;
            }
            if (count == 0 || count == maxCount)
                    return 0;
            // transition smoothing
            zM2 += 0.000000001;
            return 256* count+ (int) (255.0 * Math.log(4 / zM2) / Math.log((zRe2 + zIm2)
                 / zM2));
    }

}
```

Listing 8: MandelbrotTask class

8.3.3 MandelbrotResult class

```java
package mandelbrot;

import result.Result;

public class MandelbrotResult extends Result{
	private int id;
	private int x;
	private int y;
	private byte[] image;
	private long time;
	private long initTime;
	//The result is the portion of the Mandelbrot image, and it saves the coordinates
	    from the calculated portion of the image, and the image calculated
	public MandelbrotResult(int id, int x, int y, byte[] image, long time, long initTime
	    ) {
		super();
		this.id = id;
		this.x = x;
		this.y = y;
		this.image = image;
		this.time = time;
		this.initTime = initTime;
	}

	public int getX() {
		return x;
	}

	public int getY() {
		return y;
	}

	public byte[] getImage() {
		return image;
	}

	public int getId() {
		return id;
	}

	public long getTime() {
		return time;
	}

	public long getInitTime() {
		return initTime;
	}

}
```

Listing 9: MandelbrotResult class

8.4 P72

8.4.1 ClientP72 class

```java
package myp72;
import client.Client;
import comm.ClientPriority;

```

```
public class ClientP72 extends Client {
        private int d;
        private int numTasks;
        //Creating the Client
        public ClientP72(ClientPriority priority, int problemNumber, int version, String
            masterAddress, int masterPort, String codeServerAddress, int codeServerPort) {
                super(priority, problemNumber, version, masterAddress, masterPort,
                    codeServerAddress, codeServerPort);
        }

        public void execute() {
                long time = System.currentTimeMillis();
                d = 900000;
                numTasks = 1000;
                long ct2 = 0;
                //Creating a thread to submit tasks
                new Thread(new Runnable(){
                        @Override
                        public void run() {
                                for (int ii = 0; ii < numTasks; ii++) {
                                        //Injecting the tasks to the Coordinator - The tasks
                                            are created with fixed size 1000
                                        commit(new TaskP72(d,ii + 1, numTasks));
                                }
                        }
                }).start();
                //receiving the results and calculating the number o G.C.D.
                for (int ii = 0; ii < numTasks; ii++) {
                        ct2 += ((ResultP72) (getNextResult())).getValue();
                }
                System.out.println("ct:"+ct2);
                System.out.println("Time:␣" + (System.currentTimeMillis() - time));
        }

        public static void main(String[] args) {
                new ClientP72(ClientPriority.VERY_HIGH, 35, 35, "192.168.1.4", 0, "
                    192.168.1.4", 0).execute();
        }
}
```

Listing 10: ClientP72 class

8.4.2 TaskP72 class

```
package myp72;

import result.Result;
import tasks.Task;

public class TaskP72 extends Task {

        private int threadID;
        private int i;
        private int num_threads;
        private int ct;

        public TaskP72(int i, int ID, int threads) {
                super(ID);
                this.i = i;
                this.threadID = ID;
```

```java
			this.num_threads = threads;
		}
		//Method that is going to be executed by the worker and calculates the numbers of G.
			C.D. in the given interval
		@Override
		public void run() {
			for (int d = threadID; d <= i; d = d + num_threads) {
				int a = (d + 1) % 2;
				for (int n = 1; n < d; n = n + 1 + a) {
					if (gcd(d, n) == 1) {
						//incrementing the result
						ct++;
					}
				}
			}
		}

		@Override
		public Result getResult() {
			return new ResultP72(ct);
		}

		private final int gcd(final int a, final int b) {
			/* Eucledean Method to determine the gcd (greateast common denominator or
				highest common factor) */
			if (b == 0)
				return a;
			else
				return gcd(b, a % b);
		}
}
```

Listing 11: TaskP72 class

8.4.3 ResultP72 class

```java
package myp72;
import result.Result;

public class ResultP72 extends Result {
		//value to store the G.C.D. numbers found in the interval
		private int value;

		public ResultP72(int value) {
			this.value = value;
		}

		public int getValue() {
			return value;
		}
}
```

Listing 12: ResultP72 class