# An Investigation of Grid Computing

Submitted in partial fulfilment
of the requirements of the degree
Bachelor of Science (Honours)
of Rhodes University

Gregory Anthony Atkinson

November 6, 2006

## Abstract

Many computational problems in modern science require the manipulation of large data-sets, massive mathematical calculations or computationally intensive real-time processing, for which often the only solution is distribution over multiple computational resources. Grid computing is a tool for managing, coordinating and sharing large distributed interconnected computational, data storage and scientific resources for the purpose of solving large problems. In recent years major progress has been made in the field of Grid Computing and the paradigm shift of resource sharing and collaboration has been readily accepted by scientific, engineering and commercial communities in truly an international fashion.

This report documents investigative research into Grid computing from a review of modern literature regarding the subject, through to the deployment of a Grid computer and the use thereof. The precise definition and functionality of a Grid is explored, along with the design challenges of its deployment and use. More specifically, the ProActive Grid Computing Infrastructure is investigated and deployed in a realistic environment. An experiment is presented that benchmarks the ProActive infrastructure against the Linda Coordination Language for distributed processing, the results of which illustrate that ProActive can be a competitive solution for Grid computing while still providing high-level utilities for monitoring, security and fault tolerance. This research lead to the development of a Java-based, ProActive dependent, generic framework for simplifying application development for the Grid. The framework is discussed and demonstrated with more than satisfactory results. Finally, possible extensions to this research are presented along with a discussion of the future of Grid computing.

**Acknowledgements**

I would like to thank Amie for all her support and encouragement of my research; my supervisor, Professor George Wells, for his input, guidance and time throughout the last year; and the Computer Science department of Rhodes University for the resources that made this research possible. Acknowledgement should also be made of the departmental sponsors: Telkom, Comverse, Business Connexion, Verso, Thrip, Stortech and Tellabs.

# Contents


**1 Introduction** **6**

1.1 Intention for Researching Grid Computing . . . . . 6

1.1.1 Project Aims . . . . . 6

1.2 An Introduction to Grid Computing . . . . . 7

1.2.1 The Grid Defined . . . . . 8

1.2.1.1 Grid Computing Defined . . . . . 8

1.2.1.2 Computation and Data Clusters . . . . . 9

1.2.1.3 The Grid Identification Checklist . . . . . 9

1.2.2 Applications of Grid Computing . . . . . 10

1.2.2.1 Virtual Organisations . . . . . 10

1.2.2.2 Example Use of Grid Infrastructure . . . . . 10

1.2.2.3 Classification of Grid Software Application . . . . . 11

1.2.2.4 Benchmarking & Performance Testing . . . . . 12

1.2.3 Non-Trivial Services . . . . . 12

1.2.3.1 Interfacing & Monitoring Services . . . . . 13

1.2.3.2 Information Dissemination & Metadata Services . . . . . 13

1.2.3.3 Resource Reservation, Scheduling & Management Services . . . . 13

1.2.3.4 Security, Authentication & Authorisation Services . . . . . 13

1.2.3.5 Reliability, Check-pointing & Fault-tolerance Services . . . . . 14

1.2.3.6 Storage & Replication Services . . . . . 14

1.2.3.7 Deployment . . . . . 15

1.2.3.8 Heterogeneity . . . . . 15

1.2.3.9 Development Tools, APIs and SDKs . . . . . 15

1.2.4 Standards & Models . . . . . 16
1.2.4.1 The Layered Architecture Model . . . . . 16
1.2.4.2 Transparent Remote Objects . . . . . 17
1.2.4.3 Development Language Selection . . . . . 17
1.2.5 Current Solutions . . . . . 18
1.2.5.1 Toolkits for Grid Construction . . . . . 18
1.2.5.2 Currently Deployed Grids . . . . . 18

**2 Grid Computing with ProActive** **20**
2.1 Introduction . . . . . 20
2.2 The ProActive Grid Infrastructure . . . . . 20
2.2.1 Cluster Configuration . . . . . 21
2.2.2 Peer-to-peer Configuration . . . . . 21
2.2.3 User Interface & Tools . . . . . 23
2.2.4 Fault-Tolerance & Checkpoint Mechanisms . . . . . 24
2.2.5 Security Mechanism . . . . . 25
2.2.6 File Transfer Mechanism . . . . . 25
2.2.7 Interoperability Using Web Services . . . . . 26
2.3 The ProActive Programming Model . . . . . 27
2.3.1 Active Objects . . . . . 27
2.3.1.1 Properties and Features of ProActive Active Objects . . . . . 29

**3 Adapting the ProActive Grid** **31**
3.1 The Desired Configuration . . . . . 31
3.2 Deployment on the Linux Platform . . . . . 33
3.2.1 Configuration . . . . . 33
3.2.2 Problems Encountered . . . . . 34
3.3 Deployment on the Windows Platform . . . . . 35
3.3.1 Configuration . . . . . 35
3.3.2 Problems Encountered . . . . . 35

**4 Benchmarking ProActive against the Linda Coordination Language** **38**

4.1 Introduction . . . . . 38

4.1.1 Bioinformatics . . . . . 38

4.2 The Grid Parallel Motif Searching Application . . . . . 39

4.3 Experimental Performance Results . . . . . 42

4.3.1 Experimental Environment . . . . . 42

4.3.2 Results . . . . . 42

4.4 Application Development for the ProActive Grid . . . . . 45

**5 A Framework for Distributed Computing on the ProActive Grid** **46**

5.1 Introduction . . . . . 46

5.2 Details of the *Controller-Drone* Distributed Processing Model . . . . . 47

5.3 Possible Extensions . . . . . 49

5.4 Demonstration Application . . . . . 51

5.4.1 Introduction . . . . . 51

5.4.2 The Grid Mandelbrot Set Rendering Application . . . . . 51

5.4.3 Results . . . . . 53

5.5 Motivating the Controller-Drone Framework . . . . . 53

**6 Conclusion** **55**

6.1 Assessment of Research and Aims . . . . . 55

6.1.1 Further Investigation of the ProActive Grid Framework . . . . . 55

6.1.2 A Comparative Investigation of the Globus Toolkit . . . . . 56

**References** **58**

**A Framework** **62**

**B GridMandelbrotset** **68**

# List of Figures


1.1 The layered Grid architecture compared to the network protocol stack. Both stacks extend from the network layer to the application layer. [6] . . . . . 16

2.1 A network of hosts with some running the P2P Service [20]. . . . . 22
2.2 New peer trying to join a P2P network [20]. . . . . 22
2.3 A screen-shot of IC2D monitoring a small ProActive P2P Grid. . . . . 24
2.4 The process of calling an Active Object via SOAP. [20] . . . . . 26
2.5 A comparison between standard (passive) and Active Objects. [23] . . . . . 28
2.6 Activating an application for distribution. [23] . . . . . 28

3.1 Underlying network topology of the ProActive Grid. . . . . 31

4.1 Use case diagram for the GridPMS application. . . . . 39
4.2 Class diagram for the GridPMS application. . . . . 40
4.3 Sequence diagram for attaching to and disconnecting from the P2P ProActive Grid. . 40
4.4 Sequence diagram describing the problem domain component of the GridPMS application. . . . . 41
4.5 Speedup achieved by GridPMS and the Linda-based PMS . . . . . 43
4.6 Normalised execution time achieved by GridPMS . . . . . 44
4.7 Normalised execution time achieved by GridPMS . . . . . 44

5.1 Use case diagram illustrating the broad responsibility of the framework. . . . . 47
5.2 The framework's class diagram demonstrates the relationships between objects that comprise the framework and the attached application. . . . . 48
5.3 Sequence diagram illustrating the distribution of processing by the framework. . . . . 48
5.4 Use case diagram for the GridMandelbrotSet application. . . . . 51
5.5 Sequence diagram for the illustrating the process followed by the GridMandelbrotSet application. . . . . 52

5.6 Class diagram for the GridMandelbrotSet application. . . . . 52
5.7 A demonstration of the asynchronous nature of the communication where the results arrive out of order. . . . . 53
5.8 Zooming in and re-rendering a section of the Mandelbrot Set. . . . . 53

# Chapter 1

# Introduction

## 1.1 Intention for Researching Grid Computing

Modern science is becoming increasingly dependent on data reduction and modelling of huge data sets complex problems requiring anywhere between hours and years of computational time. Grid computing's contribution to science is as a tool that coordinates all the resources required to solve scientific problems in a cost effective manner, while encouraging the use of distributed resources and collaboration between parties with similar interests.

The original proposal for this research project was to (a) investigate the precise definition of Grid computing through an in-depth review of current literature, (b) investigate frameworks and methods for developing Grids from commodity computers and (c) port an existing bioinformatics application that searches DNA sequences to a Grid system and compare performance with the original.

The ACM Classification System (1998 Version, valid in 2006) [1] defines this research under the following two sections,

D.1.3 Concurrent Programming - Distributed Programming - Parallel Programming

I.6.7 Simulation Support Systems - Environments

where the first applies to applications utilising a Grid and the second applies to the Grid infrastructure itself.

### 1.1.1 Project Aims

A number of implementation aims were formulated to develop the best Grid for the resources and user base that was already present. Firstly, and most importantly, the Grid needed to provide an interface for application development that anyone could use without necessarily understanding the

technicalities of the Grid. Following on from this requirement is that the Grid would ideally be based on the Java programming language because the potential users of the Grid are Java literate university students and academics.

Secondly, the Grid infrastructure would be expected to make use of free CPU cycles on public workstations and ideally be portable across at least the major operating system platforms, namely Linux and Microsoft Windows, to make optimal use of the available resources. It would need to be constructed as not to disadvantage any legitimate users of the shared workstations, while still being as persistent and available as possible.

Lastly, for the sake of management and control, it would be necessary for the Grid infrastructure to be centrally manageable and if possible implement access control and security, although securing the Grid is beyond the scope of this research. The intention of these aims is to achieve maximum usability and performance on the Grid without reducing the quality of service delivered by the resources involved.

## 1.2 An Introduction to Grid Computing

The aspiration of utilising multiple distributed computing resources for a single application or project has existed for several decades. During the late 1970s work was done toward *networked operating systems,* which evolved into *distributed operating systems* in the late 1980s and early 1990s [2]. It soon became possible to coordinate highly heterogeneous systems, where *parallel distributed computing* employed parallel codes executing on distributed resources. This lead to meta-computing, and then computing on the Grid [2].

The historical development of computers, computer networks and software models reveals that the notion of Grid computing is evolutionary rather than revolutionary and it captures the existing technology trends toward networked computing [3]. Of these trends, the two most noteworthy are application evolution and network evolution. Software applications have developed from monolithic computer- or server-centric binaries to network-wide services evident in today's *Web Services* and *Service-Orientated Architectures* (SOAs) [3]. With the ease of network use and the explosive increase in network bandwidth, most modern applications benefit largely from distributed systems of storage, database and utility servers. The boundaries between computing resources have been eroded by high-speed networks creating a fabric of interwoven resources from a system of once discrete resource units [3].

The motivation for Grid computing is for the better utilisation of existing resources and for providing a platform for global collaboration in an increasing number of scientific, engineering and industrial disciplines. Modern research into these disciplines is requiring extremely high levels of processing power and petabytes of storage capacity [4], which even with ever-increasing processor speeds, disk capacities and network bandwidth, cannot be handled by discrete resources or even individual institutions. There has been an increase in the number of applications and simulations requiring on-demand

episodic access to substantial computing resources, and thus while most low-end computers (desktop PCs) are idle for approximately 30% of the day—a massive, mostly untapped resource exists in academic and commercial institutions [5]. Secondly, considering that the computational capability of microprocessors is expected to increase by a factor of 100 in the next ten years [5], the potential of idle resources is going to grow accordingly. Grid computing aims to tap this potential and produce a super-computing environment.

Grid computing might appear to be in opposition to *high-performance computers* (HPCs), but may ultimately increase, rather than decrease, demand for them. Certain computational problems will always be better suited to the tightly coupled infrastructure, low latencies and high communication bandwidths that HPCs offer [6]. HPCs will always provide a source of significant processing power, but will become just another resource managed by the Grid. Grid computing endeavours to overcome the final hurdles of complete distributed and ubiquitous resource sharing.

## 1.2.1 The Grid Defined

### 1.2.1.1 Grid Computing Defined

Grid computing has become highly popular in recent years, but due to this popularity and perhaps to its non-descriptive name, the exact definition of the Grid has been clouded [6, 7, 8]. For this reason, and with the addition of Compute Grids, Data Grids, Science Grids, Access Grids, Knowledge Grids, Bio Grids, Sensor Grids, Cluster Grids, Campus Grids, Tera Grids and Commodity Grids [7], it is necessary to provide a definition and evaluation criteria against which to measure a potential Grid infrastructure.

The term *Grid*, coined in the mid to late 1990s [6, 8], is used to denote a distributed computing infrastructure. It is often likened to that of an electrical power grid, which the Grid derived its name from, where access to computation and data should be as easy, pervasive and standard as plugging in an appliance into an outlet [8, 5]. A number of definitions have been proposed:

- *"Grid architectures are collections of computational and data storage resources linked by communication channels for shared use"* [9],
- *"A computational grid is a hardware and software infrastructure that provides dependable, consistent, pervasive and inexpensive access to high-end computational capabilities"* [5],
- *"Ensembles of distributed, heterogeneous resources"* [8],
- *a Data Grid is a specialisation and extension of "the Grid" providing storage systems and metadata management* [4] or
- *"A software system that provides uniform and location independent access to geographically and organizationally dispersed, heterogeneous resources that are persistent and supported"* [10].

These definitions each capture facets of Grid computing, but not the generalised *Grid,* which incorporates possible resources such computational capabilities, data processing, data storage and specialised sensors such as telescopes and particle accelerators executing in a heterogeneous distributed environment possible across organisational boundaries. The definition I put forward is,

> The term "the Grid" refers to a network-based computing infrastructure providing security, resource access, information and other non-trivial services that enable the controlled and coordinated sharing of resources among "virtual organisations" formed dynamically by individuals and institutions with common interests. [6]

#### 1.2.1.2 Computation and Data Clusters

A related but subtly different concept is that of a *cluster.* A cluster is a localised homogeneous network of computers connected by a high-speed local area network (LAN) designed to be used as an integrated computing or data processing resource [5, 10]. The cluster nodes may differ in configuration but not basic architecture and are usually centrally controlled by a single administrative body having complete control of the cluster [5]. A Grid architecture may be deployed on a cluster or a cluster may form part of the resources managed by the Grid.

#### 1.2.1.3 The Grid Identification Checklist

Uncertainty developed regarding the definition of a Grid [6, 7] and so the following checklist was developed to evaluate an architecture for constituting Grid or not. (This checklist was reproduced from [7].)

A Grid System...

1. ...coordinates resources that are not subject to centralised control.

   A Grid integrates and coordinates resources and users that live within different control domains—for example, the user's desktop vs. central computing; different administrative units of the same company; or different companies; and addresses the issues of security, policy, payment, membership and so forth that arise in these settings. Otherwise, we are dealing with a local management system.

2. ...uses standard, open, general-purpose protocols and interfaces.

   A Grid is built from multi-purpose protocols and interfaces that address such fundamental issues as authentication, authorisation, resource discovery and resource access. It is important that these protocols and interfaces be *standard* and *open*. Otherwise, we are dealing with an application specific system.

3. ...to deliver nontrivial qualities of service.

   A Grid allows its constituent resources to be used in a coordinated fashion to deliver various qualities of service, relating for example to response time, throughput, availability and security, and/or co-allocation of multiple resource types to meet complex user demands, so that the utility of the combined system is significantly greater than the of the sum of its parts.

## 1.2.2 Applications of Grid Computing

### 1.2.2.1 Virtual Organisations

A concept central to Grid computing is the *virtual organisation* (VO). A VO is a dynamic organisation formed by multiple institutions sharing instruments and computational resources with the purpose of collaborative problem solving [6]. Each institution has complete control over the resources it shares [6]. An example of a VO are the members of a large, international, multinational, multi-year high-energy physics collaboration, which represents an approach to computing and problem solving based on collaboration in computation- and data-rich environments [6].

The critical factor making VOs possible and motivating the move from HPCs to Grids is widespread deployment of high-speed wide-area networks [8]. The dramatic increase in inter-network connectivity has made wide-area, national and international deployment of distributed software applications feasible [8] contributing significantly to scientific research. It is the high latency, highly distributed nature of the VO that presents the incredible design challenges and requirements that Grid developers need to meet [8].

### 1.2.2.2 Example Use of Grid Infrastructure

A *National Grid*: A government usually owns a few high-end resources typically a few HPCs, petabytes of storage and hundreds of sites with thousands of smaller systems all interconnected with the fastest networks [5]. These systems are coordinated for national projects such as disaster response, national defense and long-term research and planning [5].

A *Private Grid*: Numerous technologies exist in research laboratories that could greatly benefit the medical profession that have rarely been deployed in ordinary hospitals due to their vast computational needs [5]. A Grid for a health maintenance organisation would network a number of hospitals owning a small number of high-end computers, hundreds of workstations, administrative databases, medical image archives and specialised instruments [5]. The Grid could be used to enhance desktop applications with computer-aided diagnostic and simulation applications, perform telerobotic surgery, monitor sensors and even fraud analysis of financial systems [5].

A *Scientific Grid*: The relatively new field of computational genetics, but more specifically Bioinformatics, which is a management information system for molecular biology [11]. Distributed computing is well suited to processing large data sets and [11] describes an experiment where a large set of

DNA sequences was searched using regular expressions in a distributed fashion. Their implementation utilised the TSpaces framework [12] based on the Linda coordination language [13] to distribute the computation, but this task would also be well suited to Grid deployment.

A *Scientific Grid*: A Material Science Collaboratory is an hypothetical example that comprises an international scientific group sharing a number of instruments, such as electron microscopes, particle accelerators and X-ray sources in dozens of centres, with the aim of keeping a collective data archive, specialised software and a number of supercomputers for the processing in an highly decentralised, dynamic environment. [5]. The members are able to remotely control instruments and perform online analysis utilising the full set of computational resources.

A *Public Grid*: The final community to have interest in Grid technology would be the general public or private sector of a market economy. The Grid would be utilised by the vast and varied interests of the general consumer base; various service providers such as financial modellers, graphics renderers, and interactive gamers; banks; network providers; and numerous other [5].

There are a myriad of other avenues being explored and deployed in many other fields and projects such as *global climate change*, *high-energy physics*, *radio astronomy*, *financial modelling, space exploration* and *vehicle design simulation.*

#### 1.2.2.3 Classification of Grid Software Application

The type of software application executed on the Grid will fall into one or more of the following categories.

- Loosely Coupled

  Applications in this category act on a large number of small jobs having low processing, memory and communication requirements [9]. The jobs are computationally expensive and therefore suitable for largely distributed systems interconnected by low bandwidth, high latency networks [9]. Two examples are the SETI@home [14] project discussed in section 1.2.5.2 and the DNA sequence searching application discussed in section 1.2.2.2.

- Pipelined

  Certain experiments generate more data than can be stored and consequently the data must be processed in real-time. It is then necessary that the data stream be pipelined through a computational mechanism often requiring very high processor and memory capacities (possibly more memory than can be available in a single computer), therefore needing frequent and complex inter-process communication [9]. The signal processing requirements of large radio telescopes or satellite feeds fall in this category, which is well suited to Grid computing [9].

- Tightly Synchronised

  Class 3 applications are traditionally executed on HPC systems due to their high bandwidth, heavy inter-process communication needs [9]. They may be quite data intensive and usually

have significant computational and memory requirements [9]. Examples include highly interactive climate, physics and molecular simulations employing stencil algorithms or binary cellular automata [9]. This class of application will remain in the domain of HPC systems until Grid solutions with high bandwidth and very low latency are deployed.

- Widely Distributed

  The final category pertains to applications that require little computation or memory, but are often used to update, search or synchronise distributed databases [9]. They might be related to data, but not necessarily data intensive [9].

#### 1.2.2.4 Benchmarking & Performance Testing

Information regarding the capabilities of a Grid infrastructure are important for deriving quantitative results for: the comparison of algorithms for improving the Grid software infrastructure; comparison with other Grid technologies; providing users with performance parameters describing the system's capabilities allowing them to tune their applications [9]; and developing costing and scheduling models. It is therefore necessary to define a set of performance metrics to quantify the capability and limitation one can expect from the Grid [6].

Grid benchmark efforts are divided into two main categories, *computational intensity* and *data intensity* [9]. "Computation intensity of a Grid application is defined as the amount of computational work per element of the data set(s) communicated between processes or read from a storage device. Data intensity is the reciprocal of computation intensity. [9]" Common metrics for both categories are *turnaround time*, the time between initiating a job and receiving the results, and *throughput*, the volume of data submission possible without affecting turnaround time [9].

The type of application executing on the Grid will require specific benchmark metrics for its application classification. See section 1.2.2.3 for classifications of Grid applications.

### 1.2.3 Non-Trivial Services

In its highly abstracted, dynamic, distributed, heterogeneous environment, the Grid needs to implement a number of non-trivial services for its management and coordination—this is what sets the Grid apart from ordinary parallel processing. Two critical observations, made in [8], state that the simple *client-server* approach is not adequately flexible for most VOs, but rather an architecture ranging from *client-server* to general *peer-to-peer* is required because VO members are alternately resource providers and consumers. Secondly, currently available distributed computing technologies do not fulfil the advanced sharing requirements or a decentralised mechanism for their control. This predicament has led to the development of a *Grid Software Infrastructure* [8, 4].

#### 1.2.3.1 Interfacing & Monitoring Services

Hiding complexity through the use of graphical components, interfaces and monitoring tools allows the users of the Grid to concentrate on the computational task and their research rather than the underlying complexity of the Grid architecture [15]. Abstraction is critical and the more thorough the abstraction, the better the utilisation. An excellent example of a Grid visualisation package is IC2D [16] which is part of the ProActive [17] Grid computing solution.

#### 1.2.3.2 Information Dissemination & Metadata Services

Key to the distributed system and the vast shared resources is the *Grid Information Service* (GIS) [8, 10, 9]. The GIS maintains detailed information regarding all resources available to the Grid concerning their capacities, performances, allocations and statuses in the dynamic environment [10] and it is the responsibility of the GIS to discover [8], track and index available resources keeping a live, up-to-date metadata database. The GIS also acts as the central repository for user account information and access rights. One solution demonstrated in the fields of high-energy physics and astronomy is the Chimera system [18] which defines a data schema used to established a *Virtual Data Catalog* [8]. The Globus Toolkit [19], discussed in section 1.2.5.1, implements the *Grid Resource Information Protocol* (GRIP), the *Grid Resource Registration Protocol* (GRRP), the *Grid Resource Access and Management* (GRAM) protocol and the *Grid Resource Information Service* (GRIS).

#### 1.2.3.3 Resource Reservation, Scheduling & Management Services

A service that should not be neglected is that of resource scheduling, one of the most challenging and well known Grid problems, especially for scarce resources such as large scientific instruments[10]. It is also important to co-schedule resources necessary to access the scarce resource, for example, scheduling network bandwidth and disk storage along with a particle accelerator. The scheduler should cope with applications based on systems built on-demand for limited periods of time [10] and have the ability to make intelligent decisions about selecting the optimal resources for a project. The scheduler is able to implement resource reservation, the basis on which quality of service and predictability can be guaranteed [4, 5].

#### 1.2.3.4 Security, Authentication & Authorisation Services

All relevant literature and Grid implementation recognise the need for security, a prerequisite in a highly distributed, decentralised environment where code or data could be anywhere. Security is a prime instance where complexity should be hidden from the user—ultimately the security infrastructure should be handled outside of application code. Data integrity and confidentially can be ensured by implementing cryptographic based single-sign-on authentication [6, 10] and end-to-end encrypted communication channels (provided for example by X.509 identity certificates) [10, 2]. A user should

be able to delegate his rights to another user or application, while also using local authentication utilities such as Kerberos and PKI [6]. Some Grid architectures, for example NASA's *Information Power Grid* (IPG), planned (in 2000) to deploy technologies such as IPSec and secure DNS to authenticate IP packet origin and to secure gateway and switch administration [10]. The Globus Toolkit [19] uses the *Grid Security Infrastructure* (GSI) protocol, based upon public-keys, for authentication, communication protection and authorisation to solve issues of single sign-on, delegation and integration with various local security solutions [6]

#### 1.2.3.5 Reliability, Check-pointing & Fault-tolerance Services

The Grid is by nature an unpredictable tool for several reasons: there is an absence of centralised control; the user is never sure who else is utilising a shared resource; the bandwidth available to the Grid will fluctuate depending on the type of application deployed, time of day, networks employed and so on; machine loads vary; queue lengths vary; and many other systems may fluctuate leading to variable behaviour [2]. Information concerning resource variance and reliability is required to make allocation decisions to maintain an acceptable quality of service, since it has been found that users do not only want fast execution times from their applications, but expect predictable behaviour and would be willing to sacrifice some performance for more reliable execution [2]. Components may suddenly leave the Grid due to its dynamic nature or simply lose connectivity for a multitude of reasons. Therefore, it is necessary to implement fault-tolerance and checkpoint/restart mechanisms [10] to recover from these circumstances and save weeks, if not years, of computational time guaranteeing a high quality of service.

ProActive provides two mechanisms for fault-tolerance. Firstly, *Communication Induced Checkpointing* (CIC) that takes a snapshot of the global state every TTC (Time To Checkpoint) seconds and if an error were to occur, the entire system would have to be restarted from the last checkpoint [20]. Secondly, *Pessimistic Message Logging* (PML) that logs all messages passed between objects and checkpoints individual objects [20]. While the second method requires slightly more overhead, it allows for restarting of individual objects of the application instead of the entire application [20].

#### 1.2.3.6 Storage & Replication Services

The data storage system should be abstracted from an application which should not have to be aware of data that is stored in physically separate locations on different hardware in different formats, but rather an *Application Programming Interface* (API) should remove the low-level mechanisms for data access and present a uniform view of the data [4]. It is this model of a *global file system* [10] that will cope with large, even petabyte, storage requirements.

A second area pertaining to data storage, is that of replication management and selection, a point of possible optimisation for the Grid, and it is the task of the *replication manager* [4] to manage duplicate file instances. A storage facility may provide better performance to or from a specific location or

storage space might need to be freed up for incoming data, therefore the replication manager will need to make a decision, in conjunction with its *repository* or *catalog*, as to how to produce optimal results [4].

#### 1.2.3.7 Deployment

The ease of deployment may seem to be an obvious prerequisite, but until recently, many large Grid projects have not paid much attention to this area and starting up a testbed has proven to be rather challenging. So much so, that [2] gives an example where on numerous occasions email has been sent to the project managers of several large Grid projects along the lines of "I tried to install your GridSoftwareX, but it proved rather difficult, so I decided to just write my own..." with, as one might imagine, less than satisfactory results [2].

Currently the most common method propagating the Grid software infrastructure, once the first node has been deployed, is to use something like UNIX's secure shell (SSH), secure copy (SCP) or remote copy (RCP). Thereafter updates can be propagated remotely in an automated fashion.

#### 1.2.3.8 Heterogeneity

If Grids architectures are to be allowed to span institutions and even continents, it is critical that they be constructed to operate in heterogeneous environments [15]. This involves the Grid being abstracted from the underlying hardware, software and communication platforms and itself being constructed with suitable development tools using a layer model to achieve the status of being mechanism and policy neutral [4]. Significant motivation is given for using Java [15], which is discussed in section 1.2.4.3.

#### 1.2.3.9 Development Tools, APIs and SDKs

*Application Programming Interfaces* (APIs) and *Software Development Kits* (SDKs) are pivotal because they:

- provide layered programming abstractions [8],
- enforce the correct use of protocol [8, 6],
- allow developers to create sophisticated applications in complex and dynamic execution environments [6],
- allow users to utilise those applications [6],
- provide robustness and correctness [6],
- reduce development and maintenance costs by accelerating development [6],

- enable code sharing and increase portability [6].

An example of an SDK is the Commodity Grid, or CoG, Kit [21] developed to allow existing technologies such as Java, Python and Matlab to utilise the Globus Toolkit [19]. The Java CoG Kit [22] provides access to Grid services through the Java framework.

### 1.2.4 Standards & Models

Availability of a consistent, standard service interface is fundamental [5]. The most active and effective body making progress for Grid standards is the *Global Grid Forum* (GGF) and their work on defining the *Open Grid Services Architecture* (OGSA), with strong support from companies such as IBM, Microsoft, Platform, Sun, Avaki, Entropia and United Devices, which hopefully one day will allow both open source and commercial products to intercommunicate effectively [7]. Possibly the most challenging problems are developing standards that encapsulate heterogeneity without compromising high-performance execution [5] and having the standards widely supported and implemented in the Grid architecture models.

#### 1.2.4.1 The Layered Architecture Model

The following layered model is a description of the general component classes required. Layers group similar services, which can build upon any services in a lower layer, see figure 1.1. Most of these services have been described in section 1.2.3. This section is largely a summarised version of section 4 in [6], *Grid Architecture Description*.

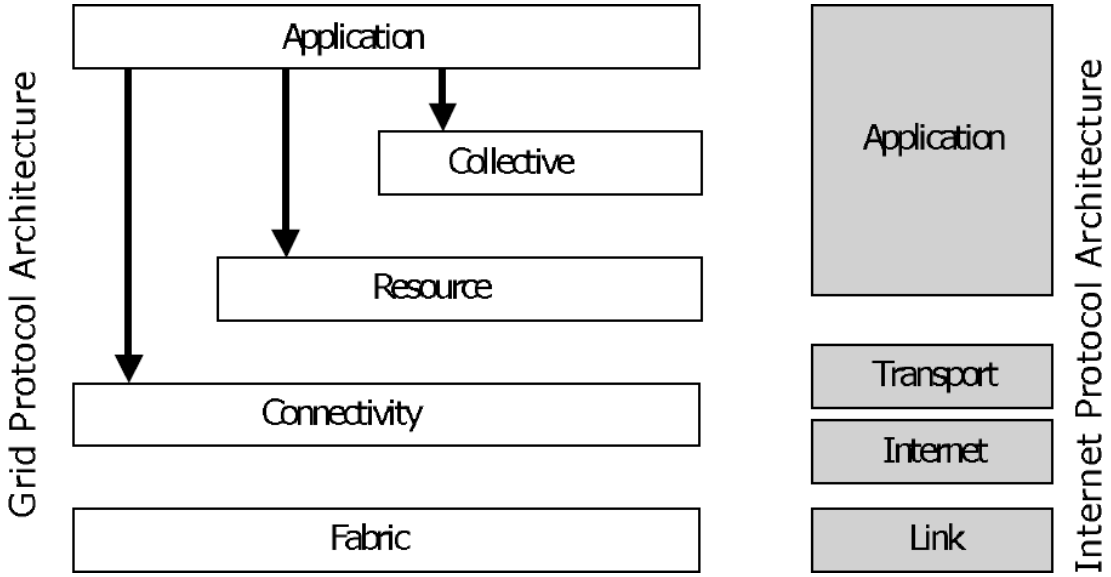



Figure 1.1: The layered Grid architecture compared to the network protocol stack. Both stacks extend from the network layer to the application layer. [6]

1. FABRIC

   The fabric layer serves the shared resources managed by the Grid. These may include distributed file systems, compute clusters, network resources, sensors, catalogs, code repositories, computational resources and storage systems.

2. CONNECTIVITY

   Communication and security protocols for all Grid transactions are defined by the connectivity layer, which enables and manages all communication between resources in the fabric layer. The security mechanisms are those as described in section 1.2.3.4.

3. RESOURCE

   Secure negotiation, initiation, monitoring, control, accounting and payment operations imposed on individual resources are defined by protocols, APIs and SDKs built into the resource layer. Information protocols derive and maintain the details regarding the structure and state of a resource and management protocols negotiate access to a shared resource.

4. COLLECTIVE

   The collective layer holds the bigger picture that is the Grid. It provides services such as directory services; co-allocation, scheduling and brokering services; monitoring and diagnostic services; data replication services; Grid-enabled programming systems; workload management systems and collaboration frameworks; software discovery services; community authorisation, accounting and payment services; and collaboratory services.

5. APPLICATIONS

   The top layer is the application layer where user applications reside. These applications are constructed mainly of the APIs defined in the lower layers calling routines for access to necessary resources and services.

#### 1.2.4.2 Transparent Remote Objects

Another model for distributed computation is the *Transparent Remote Object* (TRO) model. This approach is ideally abstracted since application objects can be deployed on remote computers creating a single, distributed application. The use of these objects is entirely transparent to the user and the only difference to application code is the way in which objects are instantiated, otherwise the objects are threaded and managed in the background and constructs such as polymorphism and inheritance still apply. The discussion of this model is continued at length in section 2.3.1.

#### 1.2.4.3 Development Language Selection

There is a significant motivation for utilising the Java programming language for distributed and Grid computing [15, 23]. Java offers a sophisticated graphical interface framework, utilities for invoking

methods on remote objects and true cross-platform portability [15]. A significant feature for distributed computing is that some JVM implementations provide access to native system threads, which when run on a multiprocessor machine, allows automatic allocation of Java threads to physical processors, breaking the barriers between monoprocessor and multiprocessor shared memory machines [23]. This feature improves performance on applications utilising concurrent mechanisms.

### 1.2.5 Current Solutions

#### 1.2.5.1 Toolkits for Grid Construction

The open source Globus Toolkit [19] is the *de facto* standard in Grid computing and has been in development for over 10 years [7]. It is a multi-institution research effort that aims to provide pervasive, dependable, and consistent access to high-performance computational resources, in an environment where users and resources may be geographical distributed [19]. The toolkit, produced by the Globus Alliance and many developers around the world, has been widely accepted for building Grid systems.

ProActive [17, 20] is another Grid middleware solution, constructed using the Java programming language and licensed under the GNU Lesser General Public License. It implements the transparent remote object methodology discussed in section 1.2.4.2. ProActive provides a comprehensive API for developing Grid applications, simple methods of deployment and both traditional and peer-to-peer mechanisms for inter-node communication. Due to Java's cross-platform nature, a Java enabled browser is sufficient to join a computer as a node to the Grid network. ProActive was extensively used throughout this research.

#### 1.2.5.2 Currently Deployed Grids

The SETI@home project allows Internet users across the globe to participate in the Search for Extraterrestrial Intelligence (SETI) by running a free application that retrieves and analyses data received by radio telescopes [14]. These telescopes listen for narrow-bandwidth radio signals emanating from outer space, signals believed not to occur naturally [14]. This is one of the oldest and most well known experiments and a prime example of highly distributed computation and data processing.

The Grid Physics Network Project is experimenting with and developing a Grid solution for scientific and engineering research requiring the collection and distribution of petabyte-scale data-sets [24]. The Grid solution in development is a *Petascale Virtual Data Grid* (PVDG) using the *Virtual Data Toolkit* (VDT) [24].

The Information Power Grid is in long-term development by NASA and aims to develop an infrastructure and services to locate, aggregate, integrate and manage its vast distributed resources [25]. Its ultimate purpose is to provide engineering and scientific communities integrated access to its resources and a substantial increase in their usability [25]. A side-effect of the project and the avail-

ability of the large computational potential will be that it can be used in exceptional circumstances such as crisis response [25].

TeraGrid is the world's biggest distributed *cyberinfrastruture* for open scientific research, with more than 102 teraflops of computing power, more than 15 petabytes of online data storage providing access to over 100 discipline-specific databases all integrated over high-performance networks [26]. It integrates high-performance computers, data resources, tools and large scientific instruments [26].

Numerous other projects include the *International Virtual Data Grid Laboratory* (iVDGL) [27], *Distributed ASCI Supercomputer* (DAS) [28] Particle Physics Data Grid [29] and Distance Computing (DisCom) Grid [30].

---

*The purpose and aims of 'An Investigation of Grid Computing' were outlined and the contribution to science by this research clearly stated. The subject and definition of Grid computing was investigated in detail through the literature published by those at the forefront of this technology. The next step in this research was to experiment with and deploy a framework that constituted a Grid system.*

# Chapter 2

# Grid Computing with ProActive

## 2.1 Introduction

The ProActive [17] Grid infrastructure is developed by the Active Objects, Semantics, Internet and Security (OASIS) project team [31] as a research initiative of The French National Institute for Research in Computer Science and Control (INRIA) [32]. It inter-operates with and builds on several official standards, namely: Web Service Exportation; HTTP Transport, JINI, OGSi[1]; SSH, RSH, RMI/SSH Tunneling; LSF, PBS, OAR, Sun Grid Engine; Globus GT2, GT3 GT4; sshGSI; NorduGrid; UNICORE and EGEE gLite. The project has also had contributions from a number of external developers, which is made possible by the complete set of source code being licensed under the GNU Lesser General Public License (LGPL) [33, 20].

The LGPL permits significant freedom to developers designing software which links libraries licensed under the LGPL. ProActive can be considered a library licensed under the LGPL and is therefore an open source product that can freely be replicated, extended and distributed along with its original copyright notice. Any software program that works with or links ProActive, that contains no portion thereof, falls outside the scope of the LGPL and can be licenced at the developer's discretion.

This chapter contains a discussion of the ProActive framework including the configurations it can be deployed in, the non-trivial services it provides that define it as a Grid infrastructure and the programming model it implements to facilitate distributed computation.

## 2.2 The ProActive Grid Infrastructure

In the previous chapter we discussed Grids, defined the Grid and outlined attributes and services that separate Grid computing from distributed processing. ProActive has been discussed as a framework for Grid computing but as yet this statement has not been qualified. The ProActive manual,

[1] Open Grid Services Infrastructure (OGSi)

*A Comprehensive Solution for Grid Computing* [20], defines ProActive as *a **Grid** Java library for **parallel**, **distributed** and **concurrent** computing, also featuring **mobility** and **security** in a uniform framework* and continues to assert that it *provides a comprehensive API allowing [the simplification of programming applications] that are distributed on **Local Area Networks** (LAN), on [a] **cluster of workstations**, **P2P desktop Grids**, or on **Internet Grids***. ProActive is a middleware solution for Grid computing that runs beneath applications designed to run on a ProActive Grid. This chapter will explore and discuss the properties of ProActive that define it as framework for Grid Computing.

### 2.2.1 Cluster Configuration

ProActive's simplest deployment is on a cluster[2], where it is explicitly started on each cluster node and each node is made aware of the the other nodes in the cluster. This allows each instance of ProActive to know where objects can be distributed to via the desired communications protocol, which may be Remote Method Invocation (RMI), Hypertext Transfer Protocol (HTTP), Remote Method Invocation over a Secure Shell connection (RMISSH), Integrated On-Board Information System (IBIS) or the Java Network Technology Protocol (JINI) [20]. Each ProActive instance on each cluster node listens on the port related to the configured communications protocol for objects and commands from other nodes. To clarify the definitions used, *virtual nodes* refer to network hosts, *nodes* to Java Virtual Machines and *Active Objects* to distributed objects.

This configuration of ProActive was not explored through this research because no dedicated cluster of computers was available for testing. A number of other Grid middleware solutions exist for utilising a cluster, but it was found through experience that the advantage ProActive's provides is its ability to deploy in a peer-to-peer configuration, provide web services and execute in a cross-platform environment.

### 2.2.2 Peer-to-peer Configuration

The ProActive peer-to-peer (P2P) infrastructure is built upon the standard cluster mode infrastructure, but with extensions to facilitate a significantly more dynamic and distributed Grid. The purpose for designing the P2P infrastructure is for maximum utilisation of an organisation or institution's spare desktop CPU cycles [20]. Very few organisations have a dedicated cluster, or are even prepared to invest in one for distributed computing, but most organisations have a large number of desktop computers that are very seldom utilised more than 60% of the day. It is significantly easier to convince an organisation to invest in a tool that will extend computational capabilities by simply better utilising resources they already own, rather than having to first acquire an additional resource base. Of course a dedicated cluster would better suit a scientific institute that could utilise the Grid for a large proportion of each day, but in the case of a university, research institute or business, a P2P Grid infrastructure sharing available user workstation's free CPU cycles is a better option.

[2]Cluster, as defined in section 1.2.1.2 and the glossary.

The advantage of using desktop workstations as Grid nodes is that nodes can dynamically join and leave the Grid as they become available and unavailable, but this presents the problem that the Grid needs to be completely decentralised, a problem solved by the ProActive P2P infrastructure's *P2PService*. The ProActive P2P infrastructure is overlayed on a dynamic network of Java Virtual Machines (JVMs), as illustrated in figure 2.1, where each peer executes a P2PService and acts as a computational node. The P2PService comprises a few *Active Objects* [20], as defined in section 2.3.1, that facilitates the integration of the peers.

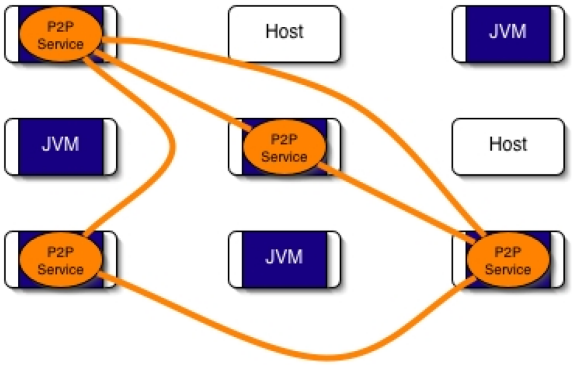



Figure 2.1: A network of hosts with some running the P2P Service [20].

When a new peer starts, its P2PService discovers and joins the P2P network. It does this by referring to a list, specified in its configuration, of peers most likely to be available and connected to the P2P network [20]. This procedure presents a bootstrapping problem for the peer that first starts, which is solved by it attempting to re-contact its list of probable peers every *Time To Update* (TTU) minutes [20]. When a peer joins a P2P network, it is introduced to all the other peers in the network so that in the event that the peer through which it connected leaves, it can still remain part of the P2P network [20]. Every TTU minutes every peer sends a heart beat to its list of acquaintances to verify their presence. Figure 2.2 illustrates a peer joining the network. Of its list of peers that could potentially be part of the ProActive P2P network two are currently connected and it joins the P2P network via those nodes, which introduce it to the rest of the network.

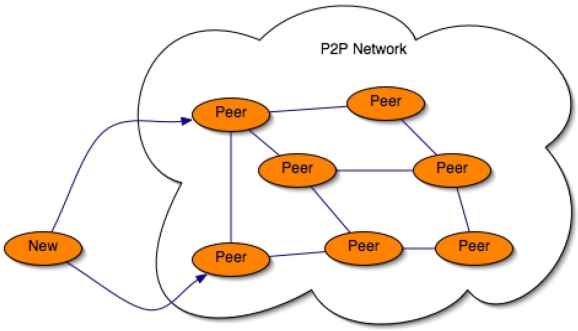



Figure 2.2: New peer trying to join a P2P network [20].

Most important to the P2P infrastructure is a resource discovery mechanism. Each peer provides an interface for enumerating the free computational nodes for deploying distributed applications and for discovering new acquaintances. It is this mechanism that nodes use to find one another and thereafter use to distribute computation.

### 2.2.3 User Interface & Tools

ProActive provides an exceptionally useful utility called *Interactive Control and Debugging of Distribution* (IC2D), which gives the user of the Grid complete control and monitoring capabilities over all deployed applications. An application can be launched from within the IC2D interface or it can be attached to an application that is already executing. IC2D works by installing a *SpyListener* Active Object on each node. The SpyListeners intercept all messages passed to and from their respective nodes and report this information, along with the state of each Active Object, back to IC2D. The user can then monitor the application in real-time during its execution.

IC2D is capable of providing the following[3]:

- Graphical visualisation of
  - Hosts, Java Virtual Machines, Active Objects
  - Topology: referencing and communications between Active Objects
  - Status of Active Objects
  - Activity migration
- Textual visualisation of
  - Messages in an ordered, causal, relative list
  - Status of Active Objects
- Control and monitoring of
  - Interactive mapping upon Active Object creation
  - Interactive destination upon Active Object migration

[3] A list based upon table 34.1 of the ProActive Manual [20]

- Dynamic change of deployment
- Drag-and-Drop migration of executing tasks

The last item in the above list hints at an extremely useful and impressive feature of IC2D. Using the IC2D interface, Active Objects can literally be dragged from one node and dropped onto another, even while they are executing without any effect on the running application, providing a manual mechanism for load balancing. This feature uses the Active Object migration facility discussed in section 2.3.1.

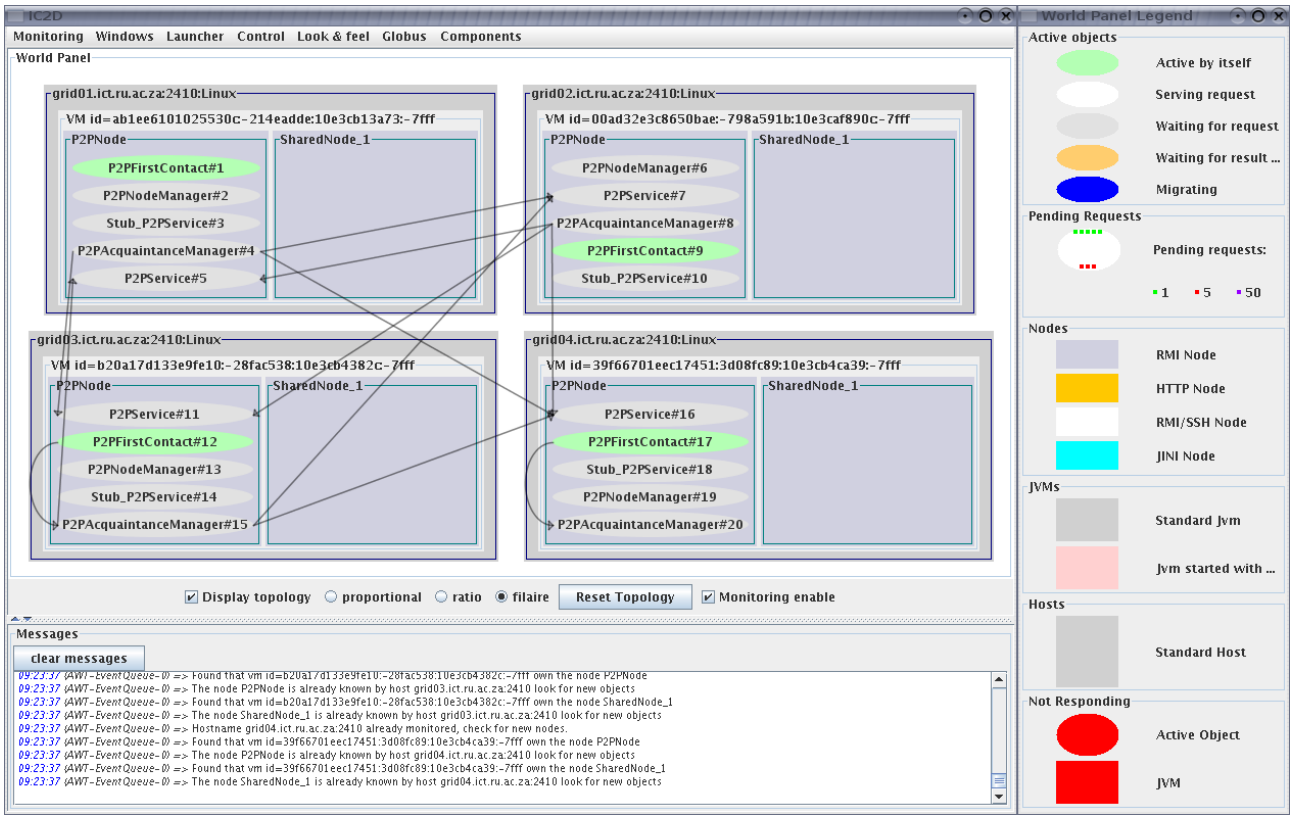



Figure 2.3: A screen-shot of IC2D monitoring a small ProActive P2P Grid.

### 2.2.4 Fault-Tolerance & Checkpoint Mechanisms

An integral part of any Grid computing framework is a mechanism for recovering from resource failure, because the greater the number of physical computers involved, the greater the probability of overall application failure becomes. There could be many reasons for resource failure when working in an highly distributed environment, such as hardware or network failure, but when working with a dynamic Grid deployed on user workstations the probability of failure becomes more likely because the user could revoke access to their workstation. The solution is to regularly checkpoint the state of the executing applications and in the event failure occurs, roll-back to the last checkpoint and

resume execution. Of course this feature adds significant computational overhead to an application and certain applications will not require fault-tolerance, but the benefit to a long running application can be great.

ProActive features a fault-tolerance server that can transparently serve executing applications using one of two possible methods. Firstly, using Communication Induced Checkpointing (CIC), where every Time To Checkpoint (TTC) seconds a global state is formed by checkpointing every Active Object in the system [20]. In the event failure occurs the whole application has to be restarted from the previous global checkpoint. The overhead introduced by this method in the failure-free case is usually low and mostly independent of the message communication rate [20]. The second method is Pessimistic Message Logging (PML) [20]. Each Active Object is individually checkpointed, including all the messages passed to it, at least every TTC seconds and no global application state is kept, therefore if failure of an object occurs, only that specific object needs to be recovered from its last checkpoint [20]. In the case of PML the overhead related to fault-tolerance checkpointing is more tightly tied to the message communication rate. In both cases, a TTC should be selected based on the predicted rate of failure, where a higher TTC value will generate less overhead in the failure-free case, but more overhead in case of failure, and vise versa.

ProActive's fault-tolerance server periodically queries each of the Active Objects in the system, normally every 10 seconds, to identify non-responsive objects. Finding a non-responsive object, it turns to the P2P network, requests an available Grid node, deploys the last checkpoint of the failed object on that node and resumes execution of that object, completely transparently to the currently executing application [20].

### 2.2.5 Security Mechanism

The Java Cryptography Extension (JCE) is the basis of the the ProActive Security Mechanism, which aims to allow dynamically deployed applications to have configurable security [20]. Security features provided include basic confidentiality, integrity and authentication and higher level policies such as migration security, hierarchical security and dynamically negotiated policies, all of which are transparently used by all running applications and can be attached to Runtimes, Virtual Nodes, Nodes and Active Objects [20]. A security policy is controlled using the Public Key Infrastructure (PKI) at three levels: by the administrator, by the resource provider and by the application [20]. ProActive is also capable of encrypting communications between Grid nodes. An investigation of Grid security was beyond the scope of this research.

### 2.2.6 File Transfer Mechanism

It is often necessary to transfer files between Grid nodes either for application specific tasks, configuration in the form of XML descriptors or even ProActive itself for on-the-fly node deployment. The ProActive file transfer model supports a number of protocols for transferring files including the

ProActive File Transfer Protocol (PFTP), Secure Copy (SCP) based on SSH, Remote Copy (RCP) base on Remote Shell (RSH) and the UNICORE [34] and NorduGrid [35] file transfer mechanisms.

The ProActive File Transfer Protocol uses a specially written API, which is part of the ProActive API, to enable file transfer during any stage of an application's execution, while the other external protocols can only be invoked during application deployment and must be configured explicitly within the XML (Extensible Markup Language) deployment descriptor [20]. The second advantage of PFTP is that as long as the Grid is operable, files can be transferred between nodes even if the external protocols are unavailable, but of course this requires ProActive to be resident on both the source and destination machines [20]. The PFTP operates by deploying two service Active Objects, one on the source and the other on the destination, and splitting the file into blocks and transferring it between the two [20].

### 2.2.7 Interoperability Using Web Services

Active Objects and ProActive components can be exported as Web Services allowing their calling and monitoring from any client written in any programming language that is Web Services enabled simply because the Web Services technology uses the XML and HTTP standards for communication [20]. *"A **web service** is a software entity, providing one or several functionalities, that can be exposed, discovered and accessed over the network. Moreover, web services technology allow heterogeneous applications to communicate and exchange data in a remotely [accessible] way.* [20]" Messages are exchanged using the Simple Object Access Protocol (SOAP) via a SOAP engine running on an HTTP web server. These messages consist of an XML encoded serialisation format used to facilitate communication over a network. The recommended configuration for ProActive based web services is to use the Apache SOAP engine running on a Jakarta Tomcat web server [20]. Figure 2.4 demonstrates the relationship and communication path between the client, SOAP interface and ProActive.

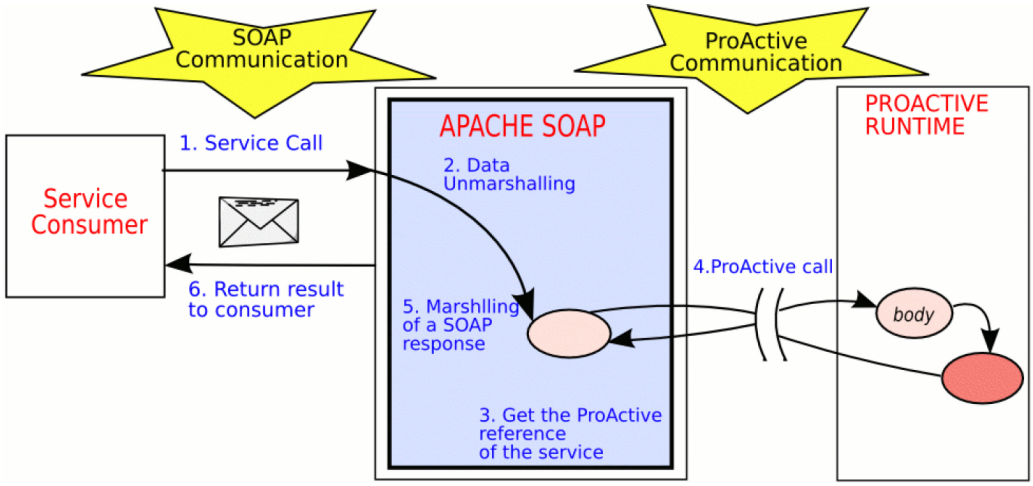



Figure 2.4: The process of calling an Active Object via SOAP. [20]

## 2.3 The ProActive Programming Model

A regular Java application only requires minor changes to enable it to operate on a ProActive Grid. These changes include basic initialisation to make the application aware of the Grid and a change in the method of instantiation for objects that require distribution, but once this has been completed, the application continues to function as a single application unaware that its constituents may physically reside on a distributed collection of computers. ProActive is completely composed of standard Java classes and therefore requires no changes to the Java Virtual Machine [36]. It provides a comprehensive Application Programming Interface (API) and graphical interface based on an Active Object pattern [36]. The API contains all the necessary tools required for creating, manipulating, grouping and synchronising Active Objects.

### 2.3.1 Active Objects

The Active Object model that ProActive employs is based on an ingenious concept called *Transparent Remote Objects* (TROs) [23, 37], mentioned in section 1.2.4.2, which builds extensively upon Java's Remote Method Invocation [38] (RMI) framework. This approach allows a single application to be deployed over multiple distributed, interconnected resources [39] and is based on the Java// (pronounced Java parallel) framework that aims to provide *seamless sequential, multi-threaded and distributed programming* [23]. The Java// library is non-intrusive, provides high-level synchronisation mechanisms, allows for the reuse of existing code and maintains constructs such as polymorphism and inheritance [23, 39]. All that is required is a standard Java Virtual Machine resulting in the framework implicitly supporting heterogeneous architectures and platforms [39]. Through using this model the application developer needs only understand the paradigm of object orientated programming to completely understand and conceptualise an application deployed on the ProActive Grid because individual objects, that comprise a autonomous application, can be instantiated on an arbitrary node of the Grid, while remaining transparently accessible to the application [39].

The TRO model extends the standard Java object in several ways giving it: **location transparency** that provides polymorphism between local and remote objects, **activity transparency** which conceals the fact that method invocations on the object occur in a separate thread and **advanced synchronisation** that allows simple and safe implementation of possibly complex synchronisation policies [23]. Transparency is achieved by using the *proxy pattern* [40], see figure 2.5, where a local object (the proxy) intercepts all communications so that the local objects do not know they are in fact communicating with remote objects [23]. A ProActive Active Object owns its own thread [20] and is actually composed of two objects, the standard object and a body [23]. The body receives and queues objects of the *MethodCall* class and executes them in an order according to the object's message acceptance policy, which by default is first-in-first-out (FIFO) [23].

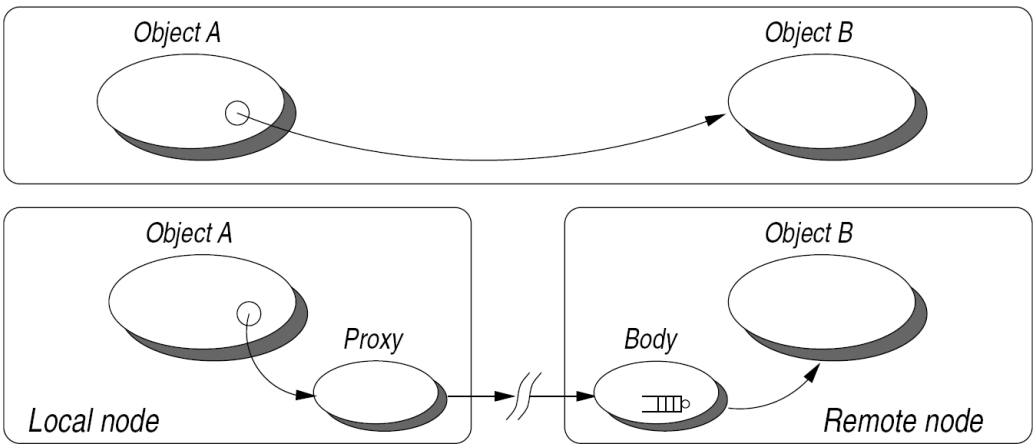



Figure 2.5: A comparison between standard (passive) and Active Objects. [23]

An application can be thought of as being divided into subsystems, each having one Active Object, and therefore one thread, and any number of passive objects [20]. Each subsystem only executes the methods within itself. All inter-subsystem communication occurs only between Active Objects and hence no passive objects are shared between subsystems. Only the Active Object of a subsystem is known to objects outside the subsystem, therefore any object, either active or passive, can reference an Active Object, but passive objects can only be referenced from within their own subsystem [20]. Using this set of rules, an ordinary application can have an appropriate set of objects made into Active Objects and hence distributed across multiple physical computers each with supporting passive objects. Figure 2.6 demonstrates the process of dividing as sequential application into a number of subsystems and distributing the subsystems across several network hosts.

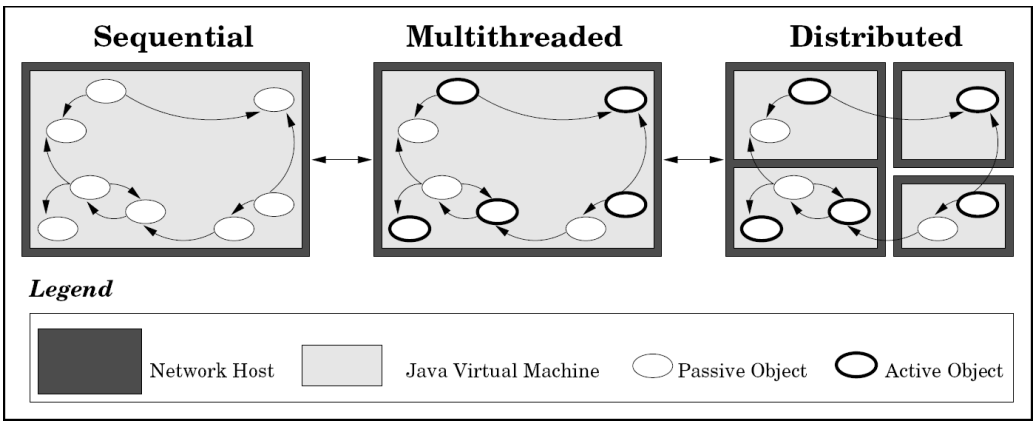



Figure 2.6: Activating an application for distribution. [23]

ProActive Active Objects can be created in one of two ways, depending on whether the application explicitly instantiates the object or if it is simply passed the object. Active Objects need to satisfy a number of restrictions: no final classes of methods are permitted, no non-public classes are permitted, the class must include an empty no-argument constructor and the serializable interface must be

implemented. It is also preferable to not to use primitive types that will force the class's objects to be treated synchronously.

- Instantiation based Active Object creation

```
Object[] constructorParams = {"text", new Integer(7)};
myClass myObj = (myClass) ProActive.newActive("myClass", constructorParams, destinationNode);
```

- Object based Active Object creation

```
passedObject = (PassedObject) ProActive.turnActive(passedObject, destinationNode);
```

#### 2.3.1.1 Properties and Features of ProActive Active Objects

**SERIALISATION**

All Active Objects, including any parameters sent to them or results received from them, need to be serialisable because all these objects need to be sent over the network.

**MIGRATION**

Active Objects can be moved from one JVM to another [36, 20] during application execution.

**GROUPING**

A typed collection of Active Objects can be created so that function calls can be performed in parallel on the collection as a whole [36].

**SYNCHRONISATION**

A function call on an Active Object returns immediately, regardless of whether that method has terminated, and a *Future* object is returned in the place of the actual object that will be the result of the function call. When the function call does terminate, it is the responsibility of that Active Object to update the *Future* it returned to the correct result [36]. The calling thread continues execution and only blocks if, (1) it requires the returned object and it is still a Future, or (2) the returned object is a primitive, hence providing *automatic continuation* [20] and wait-by-necessity, asynchronous communication. Asynchronous communication requires that Active Objects must have an empty, no-argument constructor. Synchronisation can be performed explicitly through the ProActive API if necessary.

**REACTIVE ACTIVE OBJECTS**

An Active Object can remain responsive to external events even when it is busy doing work, which can be done by implementing ProActive's *RunActive* interface [36].

---

*According to the Grid Identification Checklist, outlined in section 1.2.1.3, a Grid coordinates resources that are not subject to centralised control; uses standard, open, general-purpose protocols and interfaces; and delivers nontrivial qualities of service.*

*ProActive provides a decentralised peer-to-peer infrastructure using standard and open protocols to deliver nontrivial services such as fault-tolerance, security, monitoring, file transfer and web services integration. It can therefore be concluded that ProActive is a framework for Grid computing.*

# Chapter 3

# Adapting the ProActive Grid

## 3.1 The Desired Configuration

The resources available to this research included a VMware Virtualising server capable of running at least a dozen virtual machines depending on their memory requirements and a laboratory of just over ninety workstations used daily by university students. The strategy was to first develop a testbed of virtual machines, that are easy to create and recreate, to experimentally find a working solution for deploying a ProActive Grid. Once a working solution had been developed the next phase was to deploy on the laboratory of workstations, which consisted of a homogeneous collection of Intel Pentium 4 3.00GHz machines with 1GB of memory interconnected by a 100Mbit Ethernet Local Area Network (LAN) dual booting both Fedora Core 4 Linux and Microsoft Windows XP SP2. The network topology of the workstations and hence the Grid is illustrated by figure 3.1.

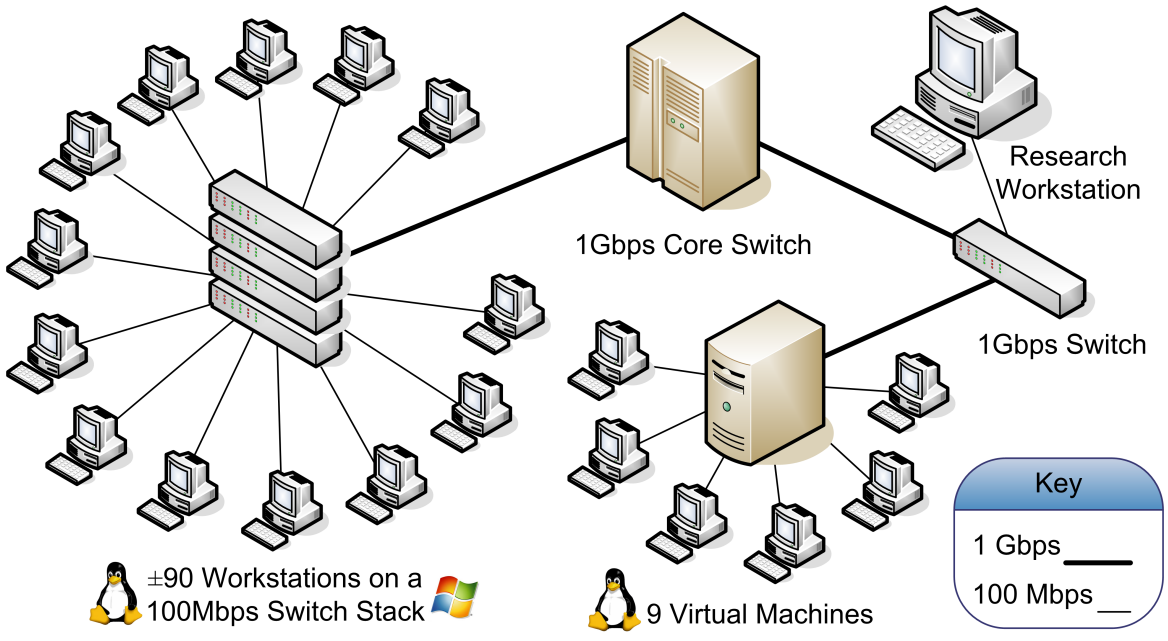



Figure 3.1: Underlying network topology of the ProActive Grid.

To best meet the requirements of the Grid, outlined in section 1.1.1, and best utilise the resources available, a number of deployment design decisions were made. Firstly, because the workstations could arbitrarily be running in either Linux or Windows at any time, both operating systems needed to be configured as Grid nodes as similarly and as simply as possible. As an aside, this was one of the primary reasons for selecting the ProActive Grid framework—its cross-platform portability. The highly dynamic nature of these workstations lead to the first design decision—using the peer-to-peer infrastructure of ProActive. The P2P infrastructure, as discussed in section 2.2.2, would be completely decentralised and consequently the Grid would not have to rely on any one node to function. It would also allow for the workstations to join and leave the Grid as users made use of them or as they were restarted into one of their two operating systems.

The second design decision was to disable and enable each Grid node on each workstation as users logged in and out respectively. This policy would maximise the use of free CPU cycles, in an already dynamic Grid, without hindering the productivity of any of the users using the workstations, an advantage when trying to convince any authority of the resources that the Grid will not adversely affect existing users. This idea, simple as it may seem, proved to be significantly more difficult to implement than originally anticipated.

Central storage of ProActive and its configuration would be an necessity and therefore the third design decision. It would be incredibly tedious, if not almost impossible, to manually alter each of ninety computers every time a change needed to be made to the Grid. Therefore, a mechanism would need to be employed to centrally manage ProActive, a facility that both the Linux and Windows operating systems provide in unique ways.

The final design decision, perhaps an obvious one, was to facilitate the best performance across all nodes and both platforms by standardising the Java environment and recompiling ProActive for that environment. Sun Microsystems' Java 5 was selected as the Java distribution of choice because of the new features it provides and because the Sun JVM is the most reliable and optimised. The Java Virtual Machine has the ability to optimally map Java threads across multiple physical processors, a desired feature for today's hyper-threading and dual core processors especially considering the highly threaded nature of ProActive's Active Objects—the Sun JVM could be relied upon for this feature. ProActive is obtainable with all its application source code and the necessary tools to compile it from source code and package it into Java binaries. It was completely rebuilt using the Sun Java 5 Development Kit version 1.5.0_06 and this version was used for both the virtualised testbed and the laboratory workstations.

The ProActive P2P infrastructure is started as a daemon service by a script provided with ProAcitve. Firstly, the P2P infrastructure is run as a daemon service so that by using a simple telnet client one can connect to the P2P service on port 2410 of the Grid node and issue commands to *start*, *restart* or *stop* the service or to *flush* the log buffer to the log file. The daemon service is particularly useful for remotely controlling the P2P grid nodes. Secondly, the provided script, a shell or batch script for Linux or Windows respectively, hides the complexity of the command line options given to a ProActive application when starting it. These scripts include a number of environment variables and include

paths. To extend the functionality of the daemon a Perl script was developed to interface the daemon to provide the same set of commands, but from the command line of the local host, and to provide further functionality such as hostname and IP address checking and execution priority selection for the Grid processes. Perl was selected as the language for the script because of its excellent and simple sockets interface, its availability for both Linux and Windows, but simply because it is the right tool for the job.

The desired configuration for the ProActive Grid would be one that incorporated all the design decisions discussed above—**a centrally controlled, decentralised peer-to-peer Grid infrastructure, deployed in a standardised Java environment, composed of dynamically available cross-platform workstations**. This would comprise a realistic setup for a Grid system deployed in an academic environment.

## 3.2 Deployment on the Linux Platform

### 3.2.1 Configuration

The configuration of ProActive on the Linux platform began on the VMware virtualisation server with nine Dapper Drake Ubuntu Linux Server installations. Ubuntu Linux Server was chosen for its minimal installation and simple configuration allowing for more time to be spent experimenting with ProActive. The various Linux distributions are similar enough to test on one and deploy on another, particularly in the case of ProActive that only has Java as a dependency. Configuration and experience is easily transferable from one distribution to another.

Each virtual machine was configured to have 128MB of memory, given its own IP address on the LAN and appeared to be an autonomous machine by anything accessing its system or services. The advantage of using virtual machines, especially when creating similar machines, is that one can be created and literally duplicated $n$ times. Ubuntu Linux Server was installed, unnecessary services disabled and Sun Microsystems' Java 5 Development Kit version 1.5.0_06 installed. An unprivileged user, *griduser*, was created and ProActive copied into its home directory. The system was configured to *rsync* (synchronise from a remote source) and start ProActive's P2P infrastructure each time the virtual machine started. A list of the hostnames of the machines in the virtual testbed was created and given to each virtual Grid node for them to find one another.

Once completely set up, the Grid computer constructed from virtual machines, functioned superbly and without incident for the duration of this research. It was continuously used in the development and testing of applications and because the Grid actually ran on one physical computer it was slow enough to analyse and debug issues of timing and asynchronous communication.

The Fedora Core 4 Linux image, on the laboratory workstations, was configured in much the same was as the Ubuntu Linux testbed, but with a few additions. Sun Microsystem's Java 5 Development Kit version 1.5.0_07 was used, an unprivileged user, *griduser*, was created and the P2P infrastructure

controlled via the Perl script mentioned above, *controlGrid-unix.pl*. The purpose of creating the griduser user was to provide a secure context for ProActive to execute within, therefore ProActive could not be used in any way to compromise the system it was on.

The only major difference between the Fedora and Ubuntu configurations was the method by which the P2P Grid infrastructure was started and stopped. On the Fedora system, ProActive was integrated with the Gnome Display Manager (GDM) that facilitates user logins. GDM includes a number of customisable scripts that are executed on *initialisation* and *pre-session*, *post-session* and *post-login*. These scripts were customised in the following ways. On *initialisation*, that is whenever the GDM becomes visible and hence while no user is logged in, ProActive is synchronised from the central location, for any configuration updates, and started. On *post-login*, that is whenever a user has successfully logged in, ProActive is stopped.

The Fedora Linux image was configured to the desired specifications. Each node executed in a secure context and joined or left the P2P network appropriately as users made use of the workstations.

### 3.2.2 Problems Encountered

One major problem was encountered, and through some in-depth investigation, solved. The problem was aggravated by the fact that ProActive was new territory and in retrospect it was a rather trivial problem with an obvious solution. This problem was discovered and solved on the Ubuntu Linux testbed and therefore was understood and avoided in the Fedora Linux deployment of ProActive.

When first started, ProActive kept binding the IP address of the loopback network device and hence would respond to other Grid nodes and inform them that the IP address on which they could be reached was 127.0.0.1. This, of course, would result in no communication between the nodes because each one would try to talk back to the P2P network over its local loopback device and hence the network traffic would remain local to the node.

This problem was discovered to be caused by a default setting in Linux where the hostname of the machine is explicitly bound to the IP address of the loopback device in the local Domain Name Service (DNS) configuration located in the */etc/hosts* file. This default is set for hosts that do not have a network interface and an external IP address, but do have client-server services on the local machine which communicate via sockets. If these services attempt to communicate by resolving the hostname to an IP address they are directed back over the loopback device and hence over the internal network, as they should be. In the event that there is an external IP address, network communications via that address from the local host are still directed back over the internal network, therefore the default setting is redundant on network hosts.

The solution to the Grid nodes binding incorrect IP addresses and not communicating was simply to remove the default setting for resolving the hostname to the internal address. The hostname would then be resolved to the external IP address of the networked host via normal DNS procedures. The interesting observation here is the method in which ProActive discovers *who* it is on the Network.

Clearly it queries Java for the hostname of the machine it is on and then asks Java to resolve the hostname to an IP address. ProActive should have a check in place to disallow being bound to the local address range, 127.0.0.0/8, because it can and should only be started once on each node and would never need to communicate via the internal network.

## 3.3 Deployment on the Windows Platform

### 3.3.1 Configuration

The Windows XP installation on each of the workstations belonged to a domain configured using Microsoft's Active Directory. All user accounts were registered on the domain and their respective home directories were centrally stored on a Microsoft Distributed File System (DFS) for access from any workstation. Therefore, it was simple to create a domain user, *griduser*, with a centralised home directory to securely encapsulate ProActive. Sun Microsystems' Java 5 Development Kit version 1.5.0_06 was installed to run ProActive and ActivePerl was installed to run the Perl interface to the P2P daemon.

A Windows service, *GridService*, was developed in Microsoft Visual C# to control the starting and stopping of the P2P infrastructure, the necessity of which will be discussed in the following section. GridService executed ProActive, via the Perl script, in the context of a local system user named *griduserService*, further encapsulating ProActive into a secure, completely read-only execution environment. When the workstation booted, GridService would start the P2P infrastructure and run it with low priority, but even when users logged in—it was not possible to configure the Windows XP environment to only execute ProActive when users were logged out.

Eventually ProActive was made to function on the Windows platform, but not to the full extent as was intended. This was no fault of ProActive's, which functioned without fault, it was simply due to the inflexibility of the Microsoft Windows platform, which through this experiment was discovered not to be a true multi-user environment.

### 3.3.2 Problems Encountered

Two major problems were encountered with the Microsoft Windows environment that significantly delayed the Grid functioning on this platform. Firstly, a network interface and IP address binding problem similar to that of the Linux platform and secondly, a serious problem of process attachment to a user's desktop.

The simpler problem of the two was again ProActive binding the incorrect IP address of the machine it was on. In the same way as before, ProActive queried the JVM for the hostname of the local machine and then the IP address corresponding to that hostname, while behind the scenes the JVM was querying Windows for the same information. When Windows is asked for the IP address of its

hostname, it returns the IP address of its first bound network interface. This situation is acceptable if the machine only has one network interface, but if it has more, one cannot guarantee which IP address corresponding to which interface will be returned. The binding order of the network interfaces in Microsoft Windows is arbitrary, but once set, persistent.

Each machine had four network interfaces, the Ethernet interface was bound second and a firewire interface was bound first. The result was that ProActive communicated its IP address, over the Ethernet network, as the firewire interface's address rather that the Ethernet address. On one particular day this had a rather interesting consequence. Of the ninety workstations, approximately half were in Linux and half in Windows. The correctly configured Linux based ProActive nodes discovered and started to communicate with the Windows based ProActive nodes, which replied with their firewire interface IP addresses, an address range that was not part of the Ethernet LAN. The Linux nodes, not recognising these IP addresses, began broadcasting ARP queries to identify which machines these address belonged to. After a few hours the number of ARP packets on the network nearly saturated the 100Mbit network and developed into a fully fledged broadcast storm. Needless to say, ProActive was disabled on the Windows platform until a solution could be found.

After some careful research, the process of changing the binding order of the network interfaces was discovered in the Microsoft online help. The binding order can be changed manually under the advanced settings in the *Network Interfaces* section of the Windows *Control Panel.* Unfortunately, there is no way this change can be scripted and each of the ninety workstations had to be altered manually. Subsequently, an IP address range check was added to the Perl script, due to this problem happening on both the Linux and Windows platforms.

The idea of starting and stopping the Grid software on each node as users logged out and logged in respectively, proved to be a significant challenge. Three methods for doing this were attempted and each one failed because of the same limitation of Microsoft Windows. As it turns out, all processes, regardless of their ownership, are attached to the *desktop* on which they are started and when the arbitrary user who owns that desktop logs out those processes are terminated. Therefore the P2P infrastructure could be successfully stopped when a user logged in, but immediately after it was started, when the user logged out, it was terminated by the user's desktop closing. This is completely different to the Unix process model where a process owned by one user has no rights over a process owned by another user, unless of course the controlling user is the superuser.

Three strategies were attempted to circumvent this limitation. Firstly an *Event Handler Dynamically Linked Library (DLL)* was developed in Microsoft Visual C++ for integration with the *Winlogon* subsystem. The DLL implemented an interface that defined several functions which were called on system startup and shutdown; user login and logout; desktop lock and unlock; and screensaver start and stop. The DLL device functioned properly and was able to correctly start the Grid software on system startup or stop in on system shutdown or user login, but as soon as it started the Grid Software before the user logged out, its processes were terminated by the desktop closing. Secondly, a similar configuration was attempted at the domain level using group policy and thirdly a Windows service was developed using Microsoft Visual C#, but in each case the outcome was the same—the logout

procedure killed the Grid processes.

Of course a solution to the problem does exist, because a number of applications and even Java applications have the ability to run as a Windows Services, but the implementation became increasingly convoluted and complex as each new possibility was explored. One hypothesis was that the initialisation chain, namely the Perl script executing the batch script executing ProActive, was the culprit and that somehow all those processes were not being held onto. But with further investigation it was discovered that this problem had been encountered many times before. Finally a possible solution was found—*Java Service Wrapper* [41] a free, open-source tool for creating Windows Services from Java applications. It recognises and claims to have overcome the problem of process termination when desktops close and would be likely solution to wrapping a Java application as a Windows Service.

---

*At this stage a number of the research aims had been achieved. The Grid computer had been deployed on both the Linux platform and Microsoft Windows platform, but with complications on the Windows platform, and was a centrally controlled, decentralised peer-to-peer Grid infrastructure, deployed in a standardised Java environment, composed of dynamically available cross-platform workstations. It utilised a public network's free CPU cycles and was persistent and available. All that remained was to develop software for the Grid, benchmark it and to determine the Grid's user-friendliness as a problem solving tool.*

# Chapter 4

# Benchmarking ProActive against the Linda Coordination Language

## 4.1 Introduction

The first application developed for this research was designed to benchmark ProActive 3.0.1 [17], against a Java[1] implementation of the Linda[2] coordination language, namely TSpaces from IBM [12]. A *Grid Parallel Motif Searching Application*, GridPMS, was designed to be as similar as possible to the Linda-based *Parallel Motif Searching Application* developed by Tim Akhurst for his master's thesis [42] and presented in his paper *Using Java and Linda for Parallel Processing in Bioinformatics for Simplicity, Power and Portability* [11]. The aim of this experiment was to explore the feasibility of using a high-level, abstracted framework such as ProActive by investigating its advantages of rapid development and simple deployment and the associated disadvantages of additional management and computational overheads.

### 4.1.1 Bioinformatics

According to [11], Bioinformatics is a relatively new discipline arising from the application of informatics techniques to biological problems, and more specifically research concerning genes and proteins. While [11] delivers a more detailed description of Bioinformatics, all that is required for the understanding of this experiment is the following. The study of genetics has its roots in DNA, a double helix shaped arrangement of four deoxyribonucleotides or bases, namely A, T, C and G—abbreviations of their initial letters [11]. A single human DNA strand is completely defined by a sequence of the bases A, T, C and G, which digitally stored consumes approximately 4GB of disk storage—the equivalent of a DVD's storage capacity. Therefore, analysis of such a vast quantity of information requires computational tools able to process, manipulate, store and search large volumes

[1] Java is a registered trademark of Sun Microsystems Inc.
[2] Linda is a registered trademark of Scientific Computing Associates.

of data in a reasonable amount of time—a problem well suited to distributed, and more specifically, Grid computing.

## 4.2 The Grid Parallel Motif Searching Application

The Grid Parallel Motif Searching Application (GridPMS), and in fact also the PMS application developed in [11], is essentially a massive, distributed, powerful regular expression matching utility able to search gigabytes of data over a large search space in just a few minutes. GridPMS was designed to be as similar as possible to the original PMS, bar the architectural differences between ProActive and Linda. The data set, used by both applications, comprised one quarter of the human genome, a 1GB text file containing DNA information as discussed in section 4.1.1. The Grid and Linda nodes used Java and Java's regular expression matching tools using a similar set of regular expressions, approximately 600, on slightly overlapping chunks of the data-set residing in each node's memory.

The biggest difference was the method by which the data-set was divided and distributed. In the case of the original PMS application the complete data-set was shared to each node via a network filesystem (NFS) mount and each node received instructions regarding which portion of the data-set to read into memory and process. GridPMS was constructed in a client-server fashion, where a *Controller*, that had access to the data-set, spawned *Drones* and passed to them the complete regular expression list and their portion of the data-set. The difference between these two approaches is that the NFS approach provides concurrent data access, while Controller-Drone pattern provides sequential data access. The sequential approach out performed the concurrent approach as more nodes were used, evident in the performance metrics acquired in figures 4.1 and 4.7.

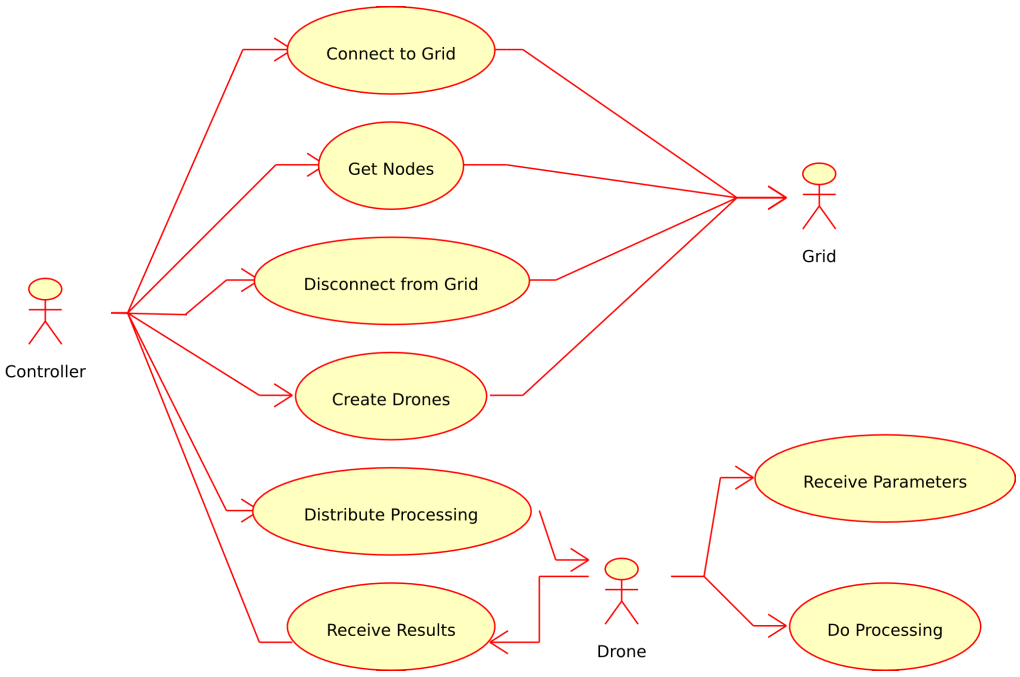



Figure 4.1: Use case diagram for the GridPMS application.

Figure 4.1 illustrates the overall functionality of the GridPMS application. A single Controller facilitates the execution of the entire application by connecting to the ProActive P2P Grid, acquiring nodes for processing, creating the Drones, distributing the processing, receiving the results and disconnecting from the Grid. The Drones simply receive the data for processing, process it and return the relevant results.

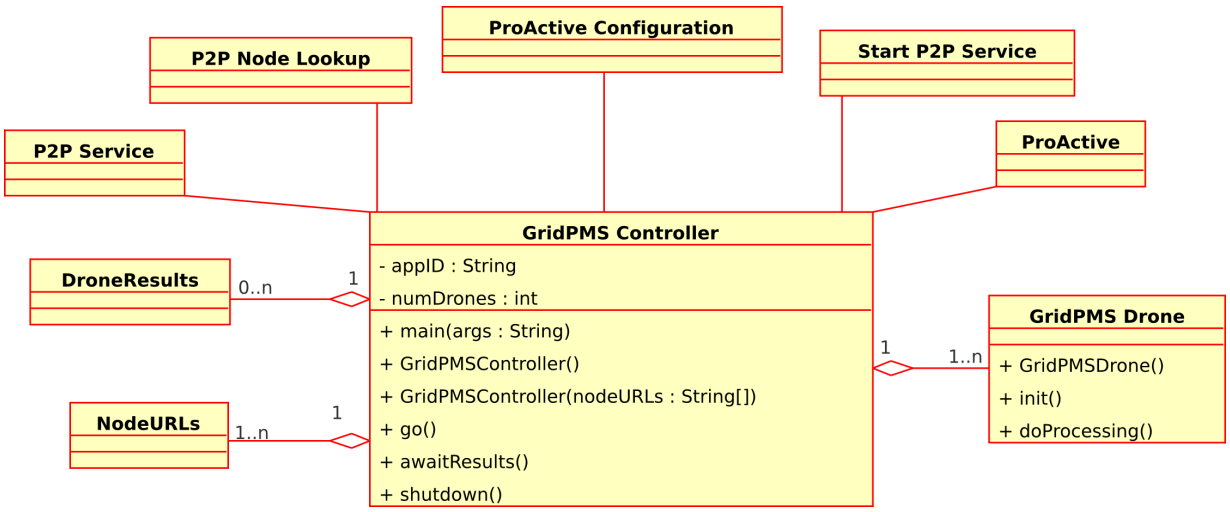



Figure 4.2: Class diagram for the GridPMS application.

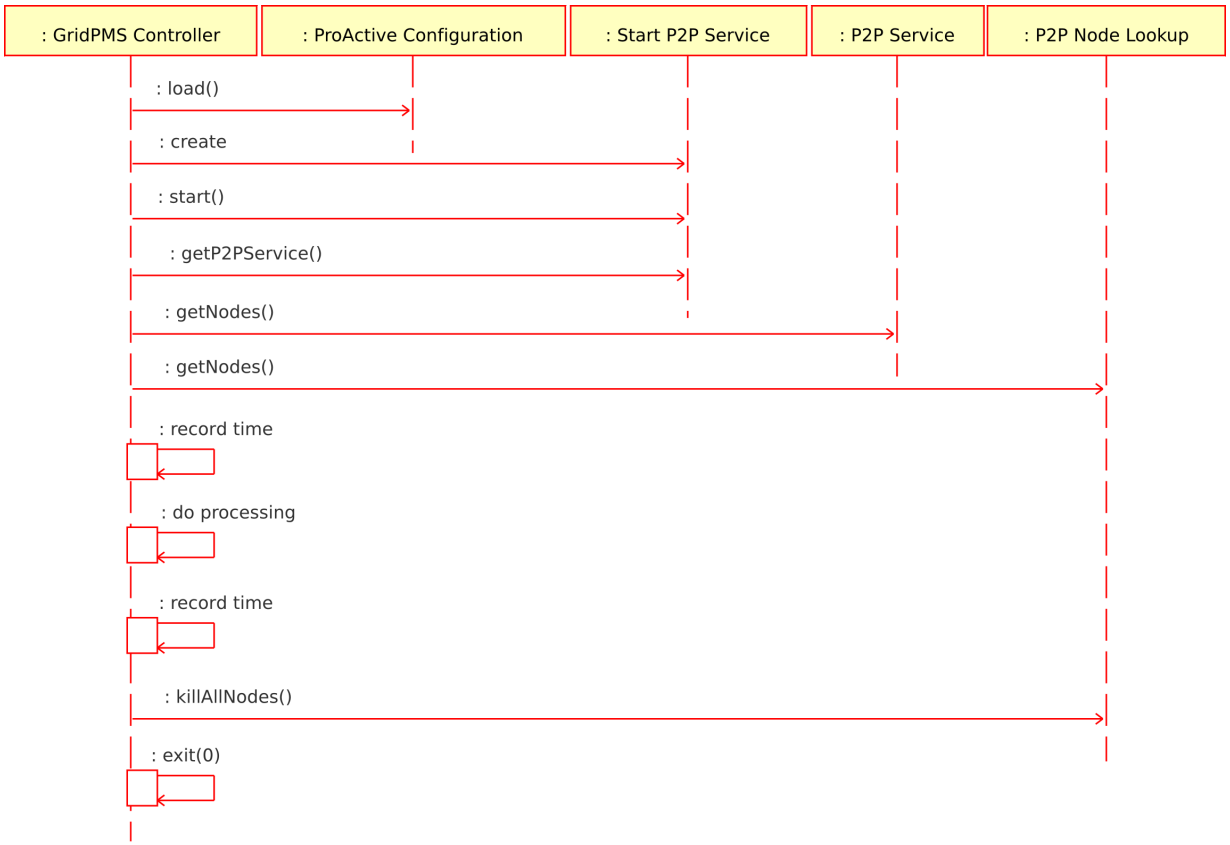



Figure 4.3: Sequence diagram for attaching to and disconnecting from the P2P ProActive Grid.

GridPMS' object model, figure 4.2, and sequence diagrams, figures 4.3 and 4.4 are relatively straight forward in their description of the system. The *StringMutableWrapper* class extends the *String* class with the necessary features, as outlined in section 2.3.1, to enable string objects to be transmitted asynchronously—the GridPMS' Drones return their results as one of these objects. The first sequence diagram captures the process of connecting to the Grid, acquiring nodes, controlling the application and disconnecting from the Grid. It can be seen that the full duration of execution is timed, including the creation of the Drones, division of the data-set, distribution of the initial parameters, processing be the Drones and retrieval of the results (start-up and shutdown times are negligible). It is important to include all these aspects to record the realistic performance of the application execution no the Grid. The second sequence diagram illustrates the problem domain component of the application, which has already been discussed in detail.

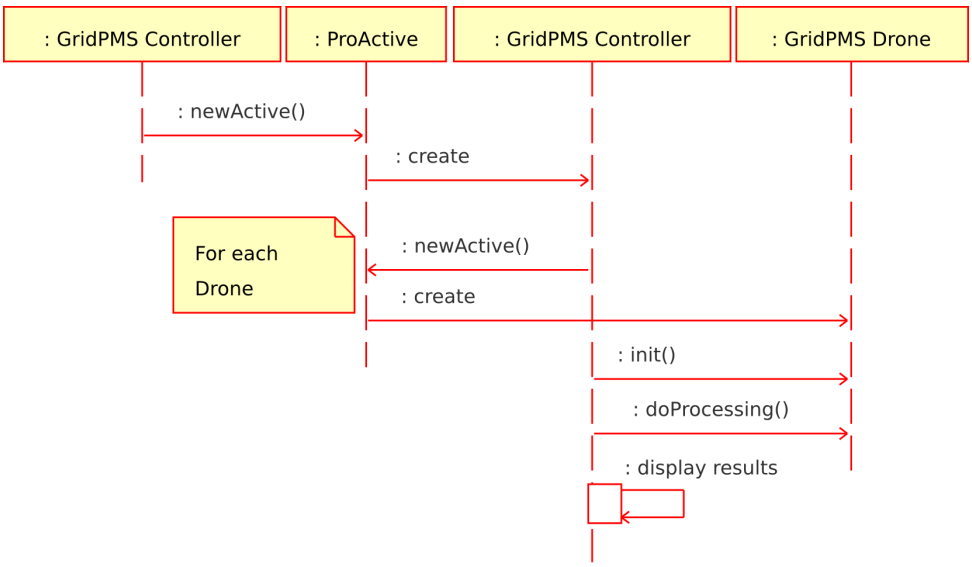



Figure 4.4: Sequence diagram describing the problem domain component of the GridPMS application.

Two observations can be made from this model Firstly, the Drones are operating in a completely diskless fashion and hence this model would function well on a cluster of diskless nodes. Secondly, from GridPMS' sequence diagram in figure 4.4 it can be seen that two Controller objects exist. The first Controller, a passive object, bootstraps a second Controller, an Active Object, and they share statically declared variables. The Controller need not be active and in the framework discussed in the following chapter the bootstrapping process was eliminated and a passive controller used. The only reason an active controller would be used is if it needed to be distributed to a Grid node, but this would only be necessary if it was doing significant processing, but all the processing should be done by the Drones, therefore a passive Controller is sufficient.

## 4.3 Experimental Performance Results

### 4.3.1 Experimental Environment

The computers used as Grid nodes were an homogeneous collection of Intel 3.0GHz Hyper-threading Pentium 4's with 1GB memory interconnected by a 100Mbps switched Ethernet local area network, each running Java 1.5.0 on Fedora Core 4 Linux and Microsoft Windows XP described in section 3.1. This differed slightly from the Intel 2.4GHz Pentium 4's with 512MB memory running Java 1.4.2 on Red Hat Linux 3.1.10 interconnected by the same network, used in [11]. Although this is a significant difference, compounded by using multiple machines, the difference is ironed out by normalising the measured experimental times, making both sets of performance results comparable.

### 4.3.2 Results

The performance increase achieved by adding nodes to the PMS is represented in figure 4.5, which is a plot of the speedup achieved versus the number of nodes employed. The speedup with $n$ nodes is measured as the ratio of the time taken by $n$ nodes to the time taken by only 1 node, hence $s(n) = \frac{t(1)}{t(n)}$. The execution time for each test run was measured for the entire duration of each run, including the time taken to distribute the data-set over the network and receive the results. The execution time when 5 nodes were used was of the order of 2.5 hours, but in each test case approximately 5 minutes was spent distributing the data-set over the network, an effect that became more apparent as more nodes were added and the execution time shortened.

Due to the complexity of the application and the large data-set, it was discovered that no less than 5 nodes could be used before the Controller exhausted its memory—a point in favour of distribution over the Grid. This occurred while the Controller was distributing the data-set to the Drones. It had to read a portion of the data-set into memory, save it as an object and send it to the Drone as part of the parameters it received to be processed. It was found that the JVM's internal handling of the data was extremely clumsy and almost three times the size of the data chunk was consumed in memory and furthermore, manual garbage collection was not able to reduce the quantity of memory used. When distributing the 1GB data-set to 5 Drones 1.1GB of memory was used, while using 50 Drones 444MB was used and 90 Drones consumed 304MB on the Controller. The ideal situation would have been to stream the data directly from the Controller to the Drones rather than first wrapping it in a object, but this can not be done with ProActive, which requires serialisable objects for communication, because Java's stream objects are not serialisable. A second option would have been to compress the data-set because the DNA data is essentially 2-bit data (representing the four bases *ACTG*) and it is quite wasteful transferring it as 8-bit or 16-bit text.

Before the results could be normalised, the execution time for only one node needed to be calculated. This was achieved by fitting the data using a non-linear least-squares regression fit to equation 4.1. The trend in figure 4.5 appeared to be an exponential increase tending to a constant value, or simply a

constant minus an exponential, which is exactly what equation 4.1 represents. To verify the accuracy of the fit, the statistical R-squared value, a measure of goodness of fit, was calculated to be 99.87%, indicating a near perfect fit to the data. This allowed $s'(n)$, the normalised speedup, to be calculated and hence compared to the results achieved in [11].

$$s'(n) = -\alpha^{\beta n - \gamma} + \frac{1}{s(1)}\alpha^{5\beta - \gamma} + 1 \tag{4.1}$$

Equation 4.1 symbols: $n$ = number of nodes; $s$ = speedup from $n$ nodes; $s'$ = normalised speedup from $n$ nodes; $\alpha$, $\beta$ and $\gamma$ are parameters solved for by the non-linear least-squares regression fit. A byproduct of the fitting to equation 4.1 is the interpretation of the constant $\frac{1}{s(1)}\alpha^{5\beta-\gamma} + 1$, which quantifies the maximum theoretical speedup achievable with infinite nodes. The maximum speedup measured was 50.5 times when using 90 nodes and the theoretical maximum was calculated to be 69.3 times.

The plot of normalised speedup, when analysed alone, can be deceiving and should be accompanied by a plot of normalised execution time verses number of nodes—figure 4.6. Figure 4.5 suggests that better performance is achieved by adding more computational units to the problem, while figure 4.6 clearly shows that at a point very little benefit is derived by adding more nodes. The optimal number of nodes for an application is very much a qualitative measure and depends entirely upon the application, its parameters and its computation to communication ratio. It can be judged from the figures 4.5 to 4.6 that using more than 15 or 20 nodes would be wasteful in the case of GridPMS with a 1GB data-set and 600 regular expressions.

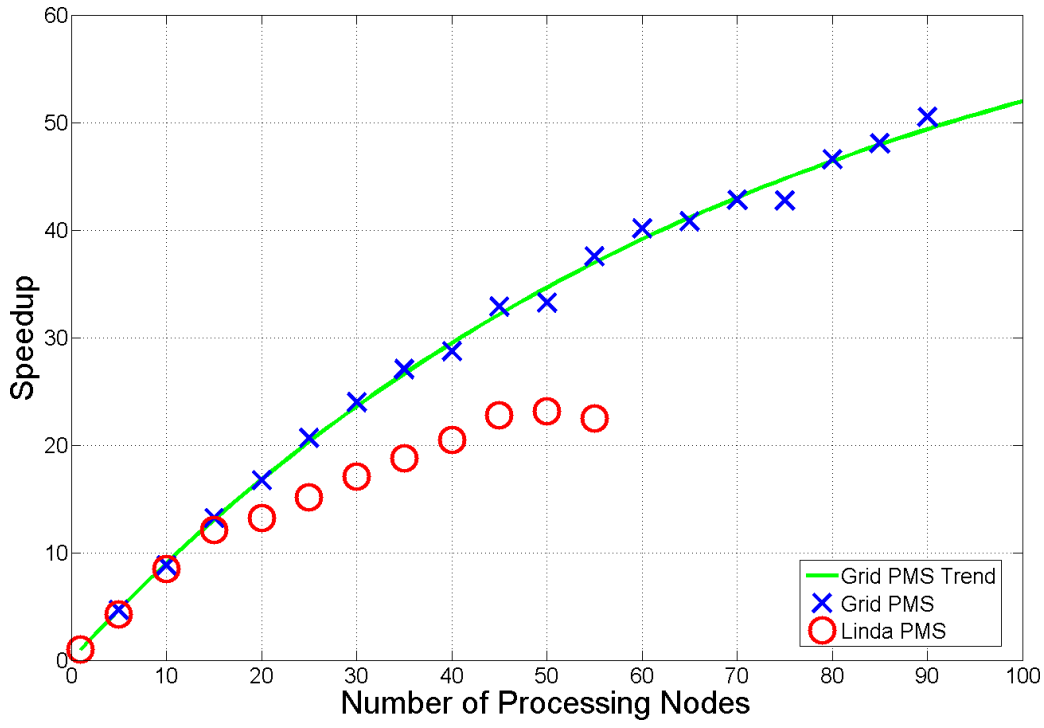
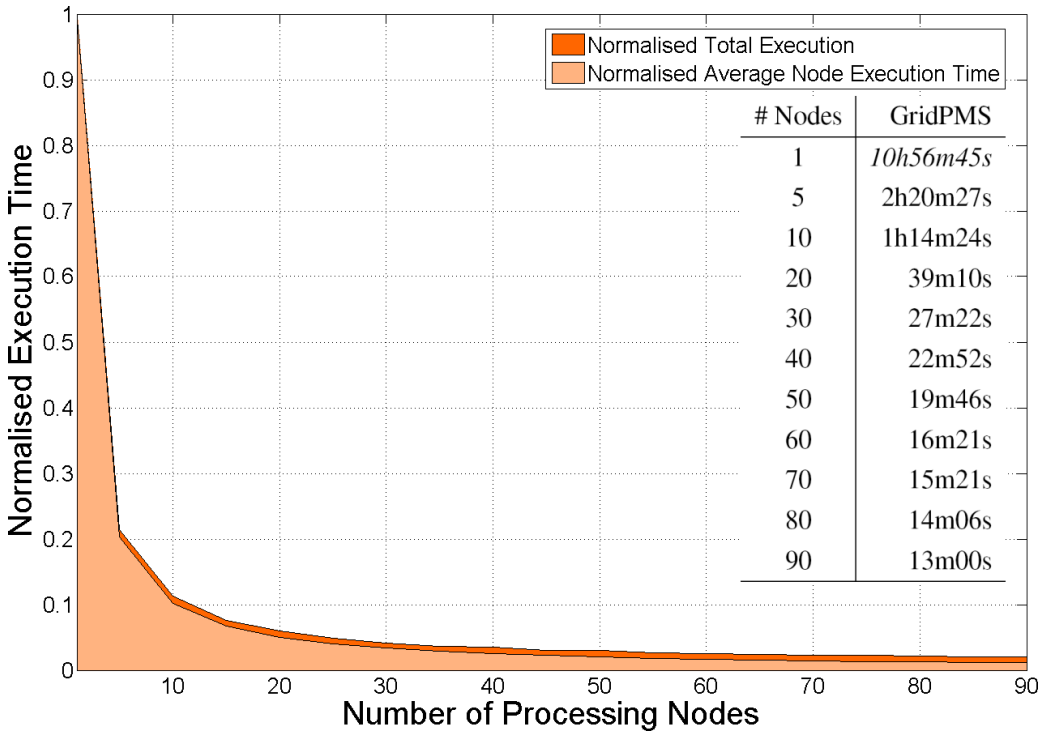



Figure 4.5: Speedup achieved by GridPMS and the Linda-based PMS




| # Nodes | GridPMS |
|---|---|
| 1 | *10h56m45s* |
| 5 | 2h20m27s |
| 10 | 1h14m24s |
| 20 | 39m10s |
| 30 | 27m22s |
| 40 | 22m52s |
| 50 | 19m46s |
| 60 | 16m21s |
| 70 | 15m21s |
| 80 | 14m06s |
| 90 | 13m00s |

Figure 4.6: Normalised execution time achieved by GridPMS

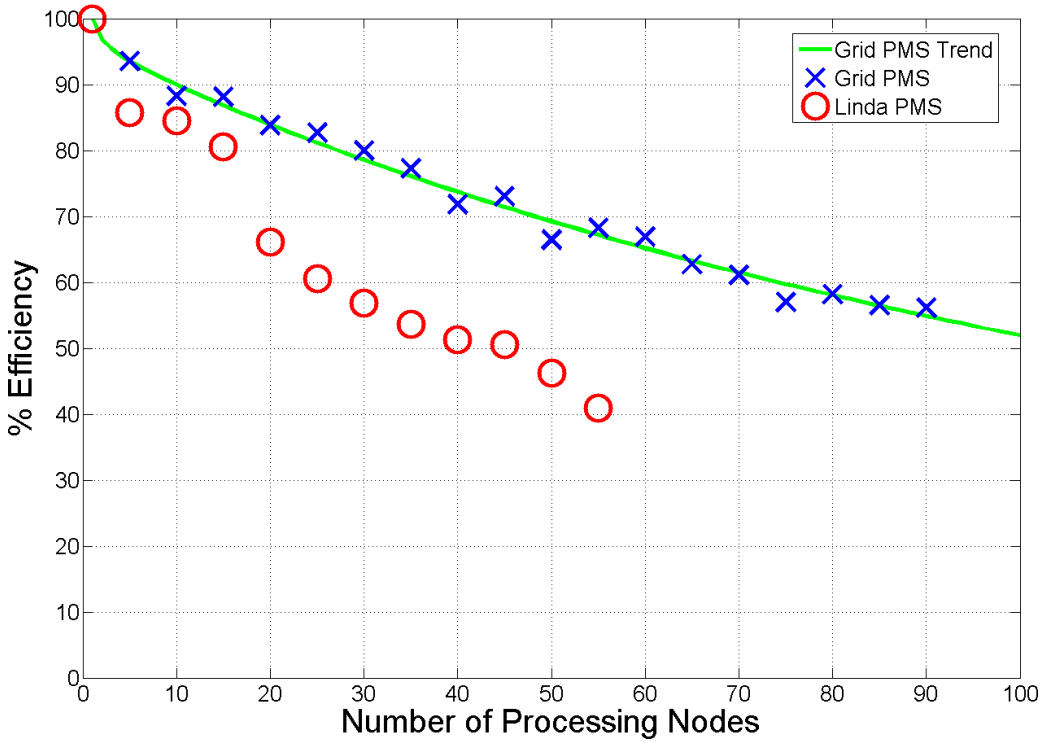



Figure 4.7: Normalised execution time achieved by GridPMS

The efficiency of a distributed application executing in parallel, calculated as $e = s'(n)/n$, quantifies how well the application utilises the distributed resources it has at its disposal. This performance measure shows the most significant difference between the Grid and Linda based implementations of

the PMS application. GridPMS is between 10% and 30% more efficient than that the Linda implementation in their use of between 5 and 45 computational nodes, evident in figure 4.7. The difference in efficiency would rather be attributed to the difference in the data distribution approach, as discussed previously, than the differences between ProActive and the Linda coordination language.

The limiting factor in the case of these PMS applications is the finite time taken to distribute the 1GB data-set to each processing node. The higher the computation to communication ratio, the more linear the speedup, the more linear the reduction in execution time and the longer the efficiency remains 100%, hence the more parallelisable the application.

## 4.4 Application Development for the ProActive Grid

One major hurdle encountered during this research and perhaps a drawback to the use of ProActive was its documentation and the availability of information and support regarding it. The ProActive Manual [20] was used as far as possible while deploying the Grid and developing software that utilised it, but unfortunately the standard of the documentation was very low and no supplementary online resources existed. For example, the grammar in parts was so poor that it made certain sections incomprehensible and certain sections referenced by the text, often source code listings, were absent. On many occasions it became necessary to resort to searching through and reading the actual Java source code to understand its functionality and requirements. At times this was frustrating, but it is the only significant criticism given of ProActive.

Bar the one criticism, developing applications using the ProActive API was a pleasure. ProActive's Active Object pattern, programming model and API made for a simple approach to solution visualisation and application development. The complete ProActive library is neatly packaged into Java Archives (JARs) and therefore easily linked into an application using its API. ProActive provides instructions [20] for using its API within the Eclipse Integrated Development Environment, an excellent environment for developing Java applications.

---

*The experiment described in this chapter was designed to gain an understanding of developing applications that utilise the ProActive Grid, solve a real world problem that requires enormous resources such as a Grid and lastly to benchmark the performance of ProActive against an existing model for distributing processing. Both the Grid and the application scaled up to 90 computers without a degradation in performance and the results achieved were comparable to those achieved when utilising the Linda Coordination Language for a similar application. The application required little extension to operate over the Grid and considering the complex services provided by ProActive for monitoring, administering and maintaining Grid applications, results were pleasing.*

# Chapter 5

# A Framework for Distributed Computing on the ProActive Grid

## 5.1 Introduction

The experience of developing and testing the GridPMS application provided valuable insight into the particularly useful *Controller-Drone* model of subdividing a computational problem and it was discovered that many parallelisable applications could be structured this way. With this in mind, a framework was developed that would greatly simplify the development process for applications that would utilise this model and completely abstract the Grid from the application.

This framework allows the developer to concentrate on the solution rather that the intricacies and internals of the Grid and its API. It handles connecting to the Grid, acquiring nodes, creating Drones, distributing the processing, receiving the results and disconnecting from the Grid. It does this while correctly and optimally threading these functions and facilitating asynchronous communications—tasks which require a good understanding of the Grid infrastructure and are tedious and difficult to implement.

The Controller-Drone framework provides a reusable generic middle tier for interfacing the Grid and separates the back-end, system interaction component of the application from the front-end, problem domain component of the application, providing benefits greater than just simplifying development. The framework's internals can be altered, optimised and extended without affecting the applications dependant on it, therefore improving the performance or capabilities of the framework does the same for all the applications that interface it. It would also be possible to leave existing applications unchanged when new versions versions of ProActive are released, while only having to adapt the framework to the new versions.

The design, internal workings, capabilities and possible extensions to the framework are discussed in this chapter. To complete the discussion, a working demonstration of the framework is presented that illustrates the simplicity of using this framework and the Controller-Drone distributed processing model.

## 5.2 Details of the *Controller-Drone* Distributed Processing Model

The application design model of the framework is similar to that of the GridPMS application discussed in the previous chapter, where the only difference lies in the fact that the framework needs to communicate with an application that supplies jobs for processing and receives results of that processing. Figure 5.1 illustrates the responsibilities of the framework; figure 5.2 the object model and the integration of the application; and figure 5.3 the control process of the framework. The same process for interfacing the ProActive Grid, as illustrated by the sequence diagram in figure 4.3, was used for the framework.

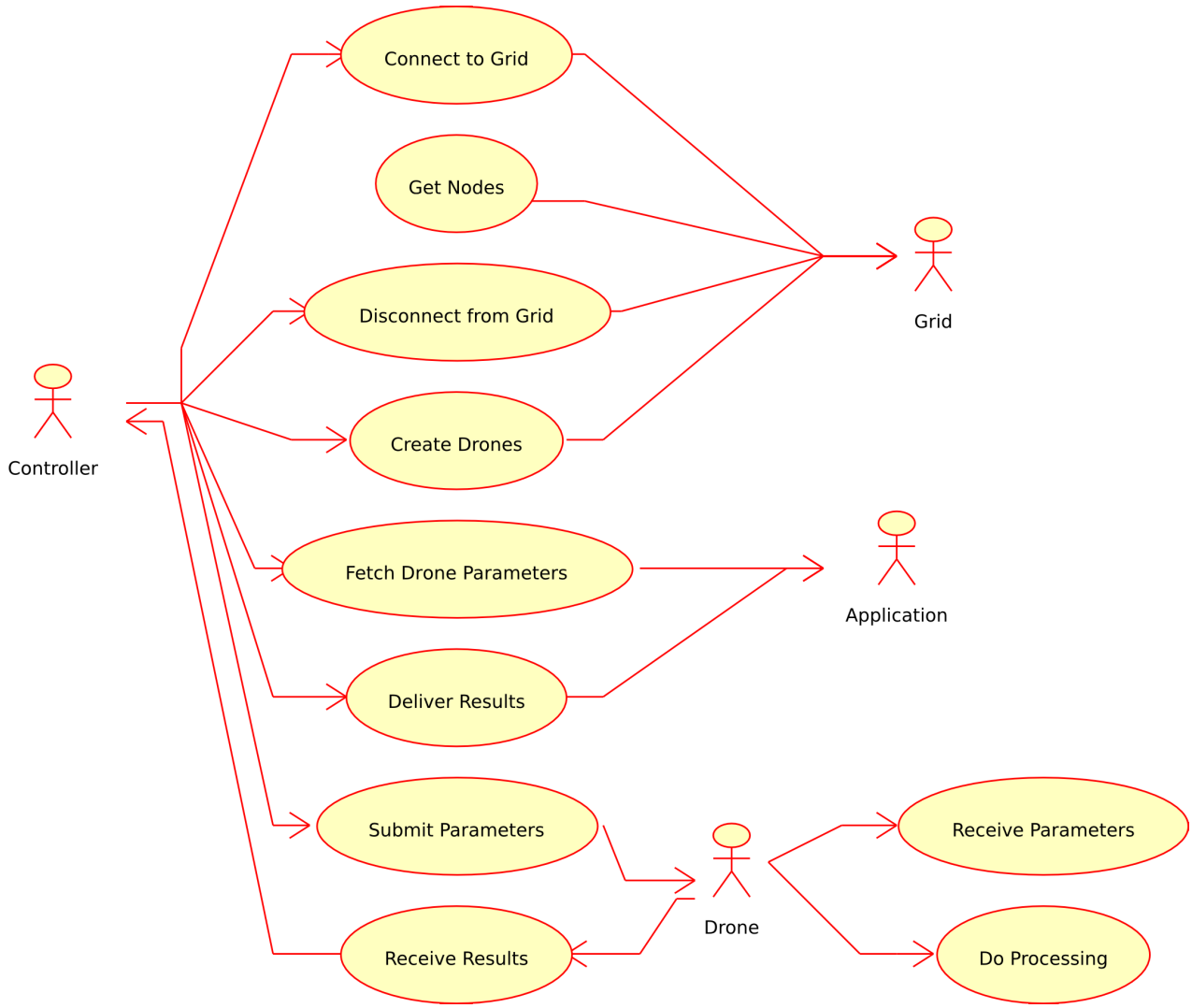



Figure 5.1: Use case diagram illustrating the broad responsibility of the framework.

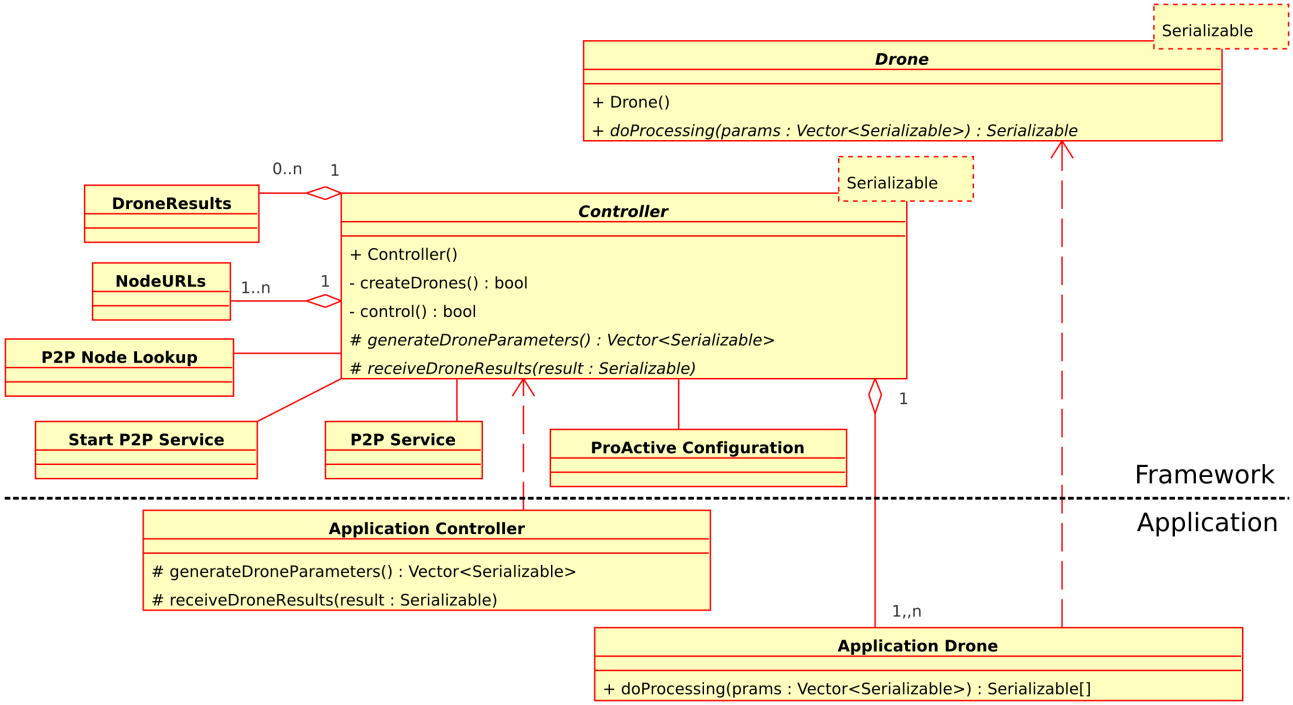



Figure 5.2: The framework's class diagram demonstrates the relationships between objects that comprise the framework and the attached application.

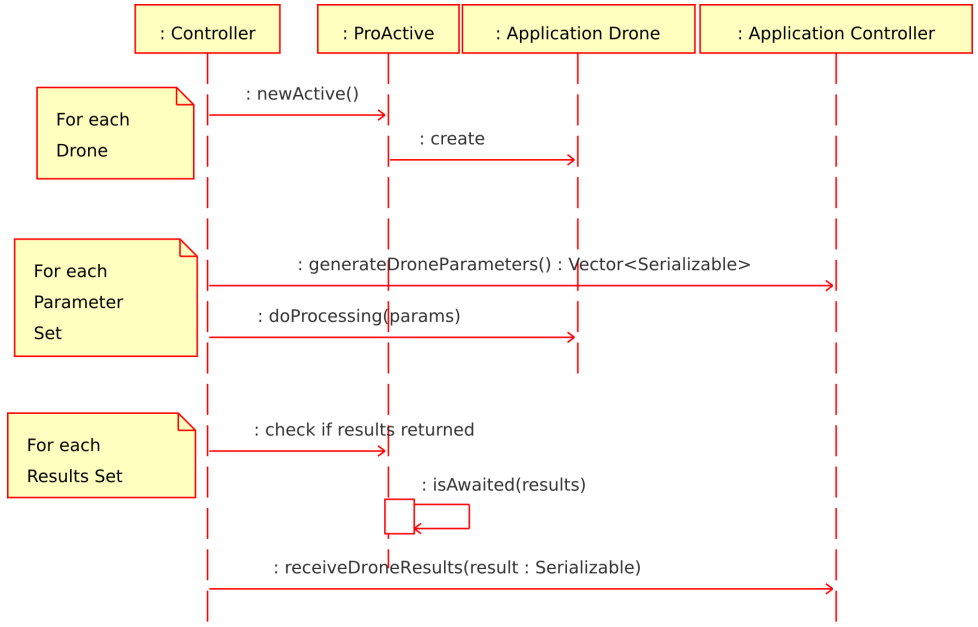



Figure 5.3: Sequence diagram illustrating the distribution of processing by the framework.

As already mentioned, the purpose of this framework is to relieve the developer from concerning himself with the particulars of the Grid. As can be seen from figure 5.2, the framework object model, all the developer needs do to execute an application on the Grid is create two classes and complete three functions within them. The first class inherits from the *Controller* class and requires the *generateDroneParameters* and the *receiveDroneResults* functions and the second class inherits from the *Drone*

class which is required to have the *doProcessing* function. The first function, *generateDroneParameters*, is called by the parent Controller when it has a free Drone. The *Vector* supplied by the function call is delivered to the *doProcessing* function of the idle Drone, which then performs the relevant processing and returns the result to the parent Controller which calls the *receiveDroneResults* function. The developer only needs to write code to generate parameters for jobs, process those parameters and process the results received, the framework handles the rest.

| | |
|---|---|
| Controller | `abstract protected Vector<Serializable> generateDroneParameters()`<br>`abstract protected void receiveDroneResults(Serializable result)` |
| Drone | `abstract public Serializable doProcessing(Vector<Serializable> params)` |

Three points are worth noting. Firstly, objects passed between the Controller and the Drones need to satisfy the constraints of Active Objects and asynchronous communications as outlined in section 2.3.1. Secondly, as in the case of GridPMS, the nodes on which the Drones reside can be diskless nodes. Finally, no bootstrapping of an active Controller is necessary, hence the framework's Controller is passive unlike GridPMS' Controller.

The Controller was designed with a number of optimising attributes. Drones are created only once, but can be reused any number of times, therefore an arbitrary number of Drones can process a greater number of jobs. The number of Drones to be used is set when the application starts and jobs are queued in the Controller until a Drone becomes available. The control algorithm for the Controller, algorithm 1, was designed to prioritise starting the Drones processing available jobs before sending available results back to the application. A source code listing can be found in appendix A.

**Algorithm 1** Algorithm implemented by the Controller.

```
while (! all jobs received || results outstanding)
while (there is a job ready && there is an available node)
* collect and send the job to an idle Drone
* start the Drone processing
endwhile
while (there is a results object awaiting)
* forward the results object to the application
endwhile
endwhile
```

## 5.3 Possible Extensions

Presently the framework only supports functionality to distribute computation as discussed above, but without any changes to its application facing interface a number of extensions could be made that would benefit all applications that utilise it. The suggested extensions are:

- Security

  ProActive's security mechanism was briefly mentioned in section 2.2.5 where the public key infrastructure can be used to encrypt network communication and provide authentication, integrity and confidentiality within the application.

- Improved Transfer of Large Amounts of Data

  When large sets of data need to be sent between objects it becomes clumsy to handle these within the Controller's memory before sending it over the network to a Drone. A preferred solution would be to have a collection of data streams between the Controller and the Drones, perhaps using the PFTP file transfer mechanism ProActive provides discussed in section 2.2.6. The streams would only be necessary for the initialisation parameters sent to the Drones because asynchronous communication is unimportant at this stage, which could be threaded if necessary—but it is imperative that the results return asynchronously.

- Fault Tolerance

  Fault tolerance was discussed at length in section 2.2.4 and would provide a significant benefit to the functionality of the framework. New Drones could be created to replace failed Drones and the application could continue uninterrupted.

- Intelligent Active Object Migration

  Building onto the idea of fault tolerance would be that of *Intelligent Active Object Migration.* Fault tolerance handles the cases where nodes uncleanly drop off the Grid and stop responding, but it adds additional overhead to the application. A mechanism that would work alongside fault tolerance and reduce its associated overhead would be that of Intelligent Active Object Migration, a mechanism that would migrate Drones from nodes that are about to leave the Grid to more reliable nodes. This would work well where public workstations that are used as nodes leave the Grid when users log into the workstations. The Drones, while still processing, could be migrated back to the controller, which could request a new node from the Grid and redeploy the active Drone, again without interrupting application execution.

- Inter-Drone Communication

  Currently the Drones have no means of communicating directly with one another, but it would be possible to make all the Drones aware of one another and able to communicate with one another. Unfortunately, this could not simply be done by passing object references from the Controller to the Drones as this would pass the entire object. The Drones would need to use ProActive's API to obtain references to the other Drones.

## 5.4 Demonstration Application

### 5.4.1 Introduction

A final discussion of the framework assumes the form of an appropriate demonstration. The demonstration is of a mathematical function that is computationally intensive to plot, requires very little communication and can be processed in a relatively short period of time, namely the Mandelbrot Set—a fractal. The Mandelbrot Set [43] is generated by using the simple equation, $z_{n+1} = z_n^2 + z_0$, where $z$ is a complex number. Each point in the complex plane is applied to the equation and the set is defined as the points for which the equation converges. When the magnitude of $z_n$ exceeds a critical limit it is mathematically guaranteed to diverge and the number of iterations needed to reach that limit is mapped to a colour on the fractal plot—points that converge are traditionally coloured black.

### 5.4.2 The Grid Mandelbrot Set Rendering Application

The Mandelbrot Set falls within the area defined by $\pm 2$ on the real axis and the $\pm 1.1$ on the imaginary axis. Zooming in to smaller and smaller regions of the Mandelbrot Set reveals greater and greater complexity with a repetitive behaviour, see figure 5.8. The *GridMandelbrotSet* application allows the user to make an arbitrary selection of the set, which it then re-renders at full size, while maintaining the aspect ratio.

The GridMandelbrotSet application is built using the Controller-Drone framework and hence its design becomes incredibly simple, as illustrated by its use case diagram in figure 5.4 and its sequence diagram in figure 5.5. It operates by dividing the plottable area into a number of strips, or jobs, that are parameterised and handed to the Controller, which in turn passes them to the Drones. The Drone computes the number of iterations required to determine the convergence or divergence of each pixel on the selected area of the complex plane, converts the iteration count of each pixel to a colour and draws the image. The final results objects, containing the iteration counts and images, are sent back to the Controller which hands them to the user interface for rendering to the screen.

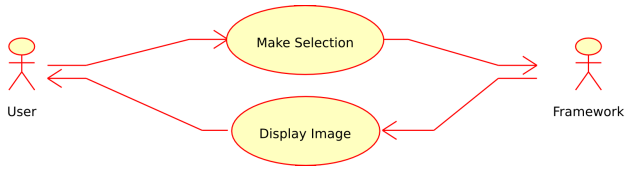



Figure 5.4: Use case diagram for the GridMandelbrotSet application.

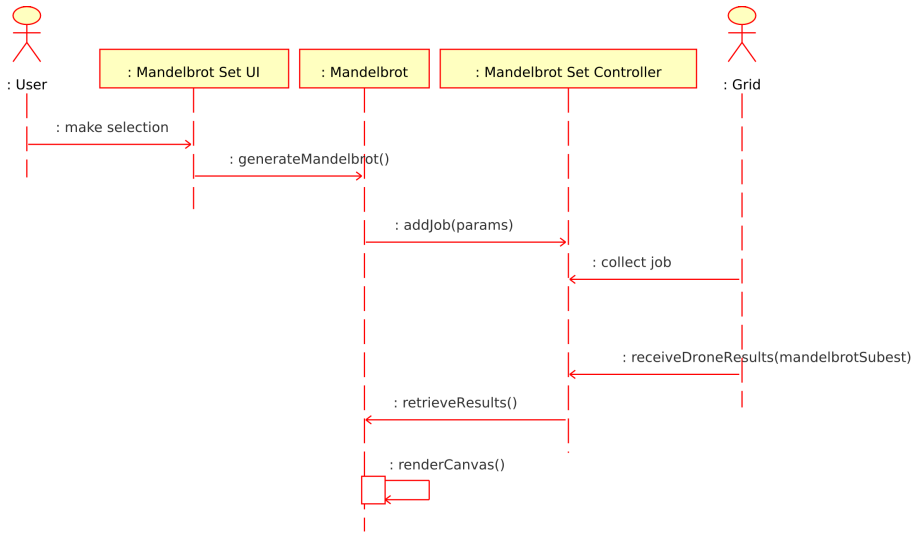



Figure 5.5: Sequence diagram for the illustrating the process followed by the GridMandelbrotSet application.

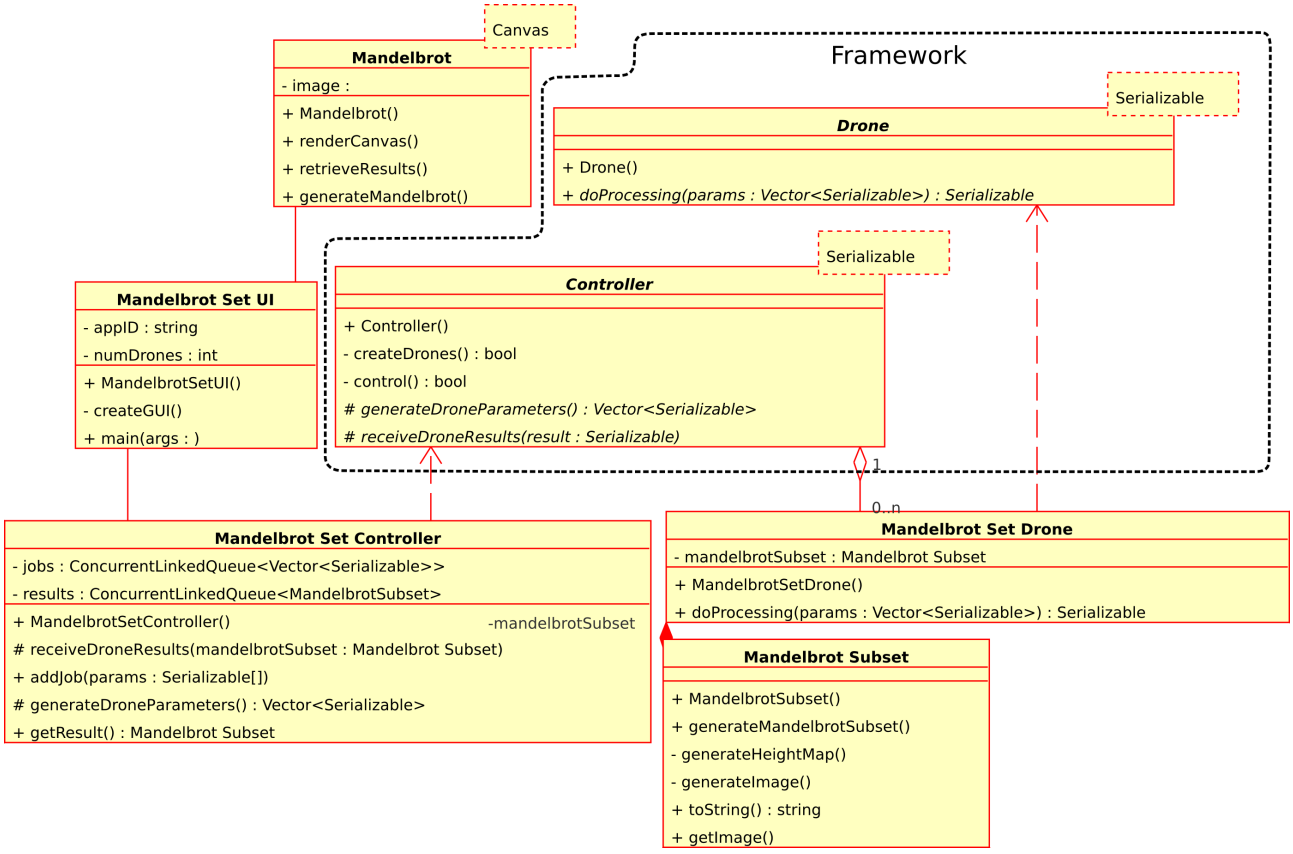



Figure 5.6: Class diagram for the GridMandelbrotSet application.

The object model, figure 5.6, demonstrates the use of the application framework. The GridMandelbrotSet application has both Controller and Drone classes which inherit from their respective parent classes, therefore the Grid is completely hidden from the context of the application. A source code listing can be found in appendix B.

### 5.4.3 Results

These images consumed only a few seconds to render on the Grid. Figure 5.7 illustrates the asynchronous nature of the returned results. The strips are distributed from the top down, but return as they are completed. The black areas are the most computationally expensive to calculate, therefore the more black contained within a strip the longer that strip takes to compute. Zooming into the Mandelbrot, demonstrated by figure 5.8, illustrates the complexity and self-reflective fractal nature of the Mandelbrot.

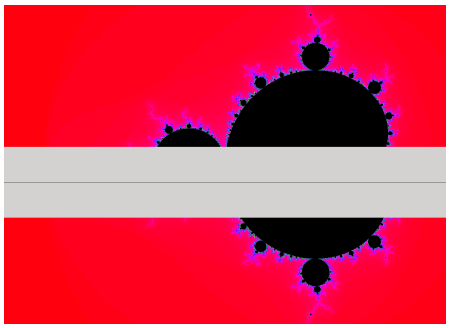


Figure 5.7: A demonstration of the asynchronous nature of the communication where the results arrive out of order.

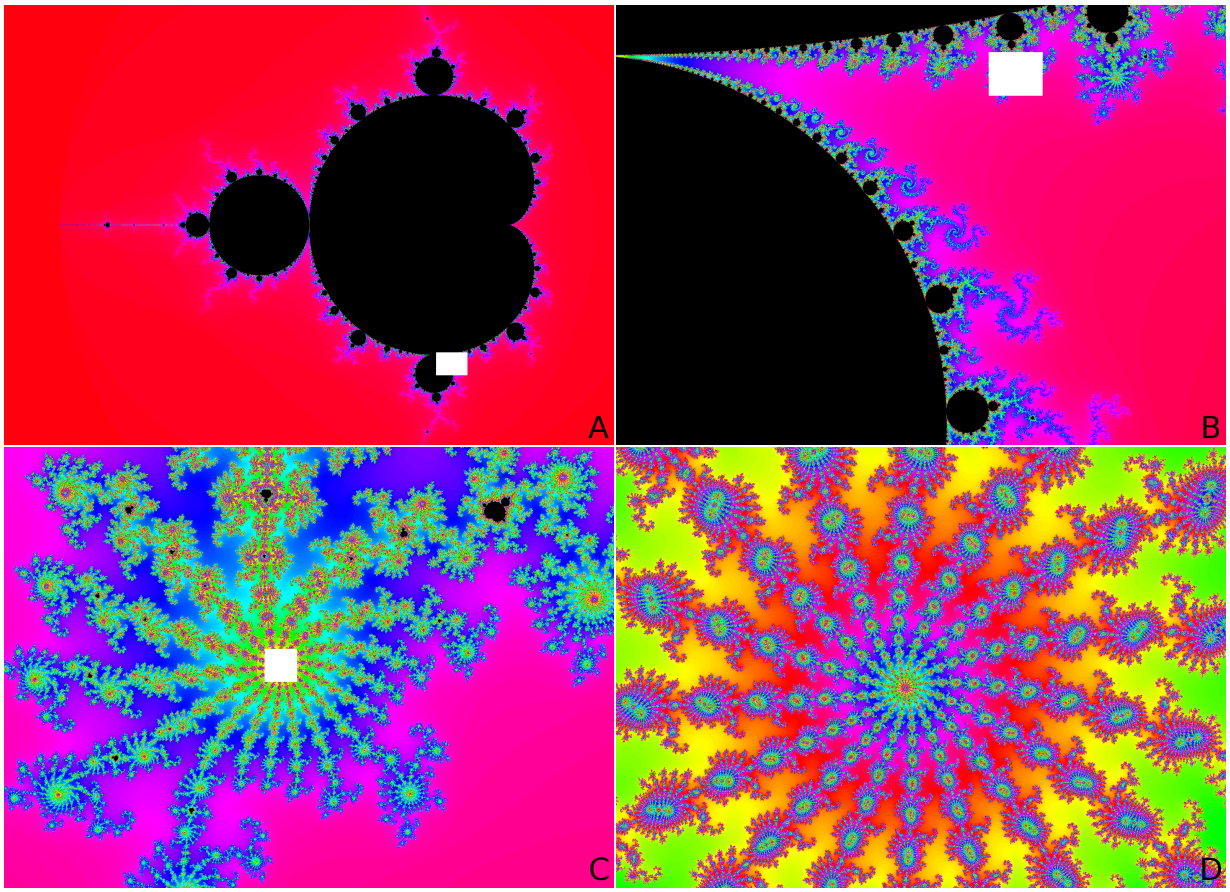



Figure 5.8: Zooming in and re-rendering a section of the Mandelbrot Set.

## 5.5 Motivating the Controller-Drone Framework

A significant amount of careful design and development went into the creation of this framework to produce a functional layer that applications developed for the ProActive Grid could be built on.

Grid related interactions such as joining the application to the Grid, requesting nodes of the Grid, correctly disposing of the used nodes and disconnecting from the Grid are functions that need to be performed in a very specific manner. As mentioned in section 4.4, it was a challenging process to engineer the aspect of the GridPMS application that interfaced the Grid because of ProActive's poor documentation—and that was before the problem domain component of the application could even be implemented. Furthermore, communication with the Drones, including related threading and waiting, is handled solely by the Controller-Drone framework, which saves the developer from difficulties related to thread safety and busy-waiting. Nodes that are requested by an application are then dedicated to that application and in the event the application does not correctly dispose of its nodes they will be unavailable to future applications. In summary, it took a number of weeks were to develop the system interaction component of the GridPMS application and a detailed understanding of the ProActive Grid—this framework completely eliminates system interaction for all applications that utilise it.

---

*A major research aim was achieved through the development of the Controller-Drone application framework—this was the aim of developing a user-friendly Grid that anyone could use to create solutions for resource hungry computational problems. The framework provides an extensible application programming interface that best utilised the Grid's capabilities and hides its complexity allowing a developer to concentrate on the application rather than the specifics of the Grid. A fractal generating demonstration application clearly illustrated the usefulness of the framework and the rapid development of Grid-based applications using the framework. The concepts and extensions discussed in this chapter have the potential to provide a springboard to a powerful, yet generic API for distributed computation.*

# Chapter 6

# Conclusion

## 6.1 Assessment of Research and Aims

In the first chapter a research proposal and a number of research aims were put forward as the desired outcomes of this project. It can be reported with confidence that proposed objectives have been achieved and the same goes for the aims, but within the constraints of the technological limitations discovered. The topic of Grid computing has been thoroughly researched and defined in section 1.2.1; an existing framework for Grid computing was investigated and deployed as a working Grid; and an application, well suited for distributed computation, was ported to the Grid and used as a performance benchmark. This was then further extended by developing a framework for simplifying application development, therefore making the Grid accessible to the user unfamiliar with the specific Grid implementation. The framework, and the Grid for a matter of fact, were both demonstrated using data intensive and processing intensive applications, both of which delivered satisfactory results.

In closing, two extensions are suggested to this research—a continuation of this research into areas that were identified during the course of this investigation that were out of scope and a comparative investigation of another popular framework for Grid computing.

### 6.1.1 Further Investigation of the ProActive Grid Framework

ProActive provides greater functionality than was utilised through this research and a number of areas were identified where further research could be conducted. Of particular interest is developing Web Services that provide and interface to the ProActive Grid. Active Objects, discussed in section 2.3.1, can be exported as a Web Service [20, 36] and deployed onto a Jakarta Tomcat web server as described in section 2.2.7. This mechanism could make the Grid significantly more usable by providing the ability to create client applications in any Web Service enabled programming language.

Secondly, the extensions suggested to the development framework, outlined in section 5.3, could be completed. This would integrate security, fault tolerance, improved file transfer and improved communication within the application, thereby using the Grid to create a robust application environment.

### 6.1.2 A Comparative Investigation of the Globus Toolkit

The Globus Toolkit, a highly developed infrastructure for Grid computing, mentioned in section 1.2.5.1, would form the basis of an excellent comparison to this research. A Globus-based Grid could be deployed and a similar set of applications could be developed for testing and benchmarking against those presented in this report. Furthermore, the Commodity Grid (CoG) Kits [22, 44], that provide an interface for the Java, Python and Perl programming languages to the C/C++ based Globus infrastructure, could be investigated.

---

*The Grid is still largely a developing technology, with huge potential and a long way still to go, but some research institutes and financial organisations are starting to implement this technology. It is not the next generation Internet, nor an unlimited source of resources, but will rather be an Internet enabled tool for maximising use of existing resources [6]. When the full power of the Grid is realised it will provide awesome possibilities to the scientific community and commercial sector making as yet unrivaled computational power and resource access available to the general scientist and citizen for application as wide as imagination allows.*

*In summary, the achievement of this research was a clarification of Grid computing, the deployment of a functional high-performance Grid capable of challenging other distributed infrastructures and the development of an application framework enabling rapid implementation of problem solving applications requiring only a basic understanding of the Grid.*

# Glossary

*Grid* A network-based computing infrastructure providing security, resource access, information and other non-trivial services that enable the controlled and coordinated sharing of resources among "virtual organisations" formed dynamically by individuals and institutions with common interests.

*Cluster* A localised, homogeneous network of computers connected by a high-speed local area network (LAN) designed to be used as an integrated computing or data processing resource.

*ProActive* A Grid Java library for parallel, distributed and concurrent computing, also featuring mobility and security in a uniform framework.

*GridPMS* A massive, distributed, powerful regular expression matching utility able to search gigabytes of data over a large search space.

*Virtual Organisation* A dynamic organisation formed by multiple institutions sharing instruments and computational resources with the purpose of collaborative problem solving.

*Bioinformatics* A scientific discipline where informatics techniques are applied to biological problems.

# References


[1] The ACM Computing Classification System [1998 Version]. [Online]. Available: http://www.acm.org/class/1998/

[2] J. Schopf and B. Nitzberg, "Grids: Top Ten Questions," *Scientific Programming, special issue on Grid Computing*, vol. 10, no. 2, pp. 103–111, August 2002. [Online]. Available: http://www.globus.org/alliance/publications/papers/topten.final.pdf

[3] P. Strong, "Enterprise Grid Computing," *Queue*, vol. 3, no. 6, pp. 50–59, 2005. [Online]. Available: http://doi.acm.org/10.1145/1080862.1080877

[4] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke, "The Data Grid: Towards an Architecture for the Distributed Management and Analysis of Large Scientific Datasets," *Journal of Network and Computer Applications*, vol. 23, pp. 187–200, 2001. [Online]. Available: http://www.globus.org/alliance/publications/papers/JNCApaper.pdf

[5] I. Foster, *Computational Grids*, C. Kesselman, Ed. Morgan-Kaufman, 1999, Chapter 2 of "The Grid: Blueprint for a New Computing Infrastructure". [Online]. Available: http://www.globus.org/alliance/publications/papers/chapter2.pdf

[6] I. Foster, C. Kesselman, and S. Tuecke, "The anatomy of the grid: Enabling scalable virtual organizations," *International J. Supercomputer Applications*, vol. 15, no. 3, 2001. [Online]. Available: http://www.globus.org/alliance/publications/papers/anatomy.pdf

[7] I. Foster, "What is the Grid? A Three Point Checklist," *GRIDToday*, July 2002. [Online]. Available: http://www-fp.mcs.anl.gov/~foster/Articles/WhatIsTheGrid.pdf

[8] S. Rajsbaum, "ACM SIGACT News Distributed Computing Column 8," *SIGACT News*, vol. 33, no. 3, pp. 50–70, 2002. [Online]. Available: http://doi.acm.org/10.1145/582475.582485

[9] A. Snavely, G. Chun, H. Casanova, R. F. V. der Wijngaart, and M. A. Frumkin, "Benchmarks for Grid Computing: A Review of Ongoing Efforts and Future Directions," *SIGMETRICS Perform. Eval. Rev.*, vol. 30, no. 4, pp. 27–32, 2003. [Online]. Available: http://doi.acm.org/10.1145/773056.773062

[10] W. E. Johnston, D. Gannon, B. Nitzberg, L. A. Tanner, B. Thigpen, and A. Woo, "Computing and Data Grids for Science and Engineering." in *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*. Washington, DC, USA: IEEE Computer Society, 2000, p. 52. [Online]. Available: http://www.supercomp.org/sc2000/Proceedings/techpapr/papers/pap253.pdf

[11] G. Wells and T. Akhurst, "Using Java and Linda for Parallel Processing in Bioinformatics for Simplicity, Power and Portability," in *Proc. IPS-USA-2005*, Cambridge, MA, USA, June 2005.

[12] P. Wyckoff, S. W. McLaughry, T. J. Lehman, and D. A. Ford, "'TSpaces' IBM Systems Journal," vol. 37, no. 3, pp. 454–474, 1998.

[13] D. Gelernter, "Generative communication in Linda," *ACM Trans. Programming Languages and Systems*, vol. 7, no. 1, pp. 80–112, January 1985.

[14] University of California. SETI@home. [Online]. Available: http://setiathome.berkeley.edu/

[15] V. Getov, G. von Laszewski, M. Philippsen, and I. Foster, "Multiparadigm Communications in Java for Grid Computing." *Commun. ACM*, vol. 44, no. 10, pp. 118–125, 2001. [Online]. Available: http://doi.acm.org/10.1145/383845.383872

[16] F. Baude, D. Caromel, L. Mestre, F. Huet, and J. Vayssière, "Interactive and descriptor-based deployment of object-oriented grid applications," in *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*. Edinburgh, Scotland: IEEE Computer Society, July 2002, pp. 93–102.

[17] Inria Sophia Antipolis. ProActive. [Online]. Available: http://www-sop.inria.fr/oasis/ProActive/

[18] "Chimera: A Virtual Data system for Representing, Querying, and Automating Data Derivation," in *14th International Conference on Scientific and Statistical Database Management, Edinburgh*, July 2002.

[19] I. Foster and C. Kesselman, "Globus: A Metacomputing Infrastructure Toolkit," *Intl J. Supercomputer Applications*, vol. 11, no. 2, pp. 115–128, 1997. [Online]. Available: ftp://ftp.globus.org/pub/globus/papers/globus.pdf

[20] OASIS Research Team, *ProActive: A Comprehensive Solution for Grid Computing*, v3.1 ed., April 2006, the ProActive Manual. [Online]. Available: http://www-sop.inria.fr/oasis/proactive/doc/api/org/objectweb/proactive/doc-files/index.html

[21] G. von Laszewski. Commodity Grid Kits. [Online]. Available: http://wiki.cogkit.org/

[22] G. Laszewski, I. Foster, J. Gawor, and P. Lane, "A java commodity grid kit," 2001. [Online]. Available: citeseer.ist.psu.edu/laszewski00java.html

[23] D. Caromel, W. Klauser, and J. Vayssiere, "Towards Seamless Computing and Metacomputing in Java," in *Concurrency Practice and Experience*, G. C. Fox, Ed., vol. 10, no. 11–13. Wiley & Sons, Ltd., September-November 1998, pp. 1043–1061. [Online]. Available: http://www-sop.inria.fr/oasis/proactive/doc/javallCPE.ps

[24] US National Science Foundation. GriPhyN. [Online]. Available: http://www.griphyn.org/

[25] W. E. Johnston, D. Gannon, and B. Nitzberg, "Grids as Production Computing Environments: The Engineering Aspects of NASA's Information Power Grid," in *HPDC*, 1999. [Online]. Available: citeseer.ist.psu.edu/johnston99grids.html

[26] US National Science Foundation. TeraGrid. [Online]. Available: http://www.teragrid.org/

[27] International Virtual Data Grid Laboratory. [Online]. Available: http://www.ivdgl.org/

[28] Vrije Universiteit Amsterdam, University of Amsterdam, Delft University of Technology, University of Leiden and University of Utrecht. The Distributed ASCI Supercomputer (DAS). [Online]. Available: http://www.cs.vu.nl/~bal/das.html

[29] Particle Physics Data Grid. [Online]. Available: http://www.ppdg.net/

[30] Sandia National Laboratories. Distance Computing (DisCom) Grid. [Online]. Available: http://www.cs.sandia.gov/discom/

[31] INRIA. Active Objects, Semantics, Internet and Security (OASIS) Project Neam). [Online]. Available: http://www.inria.fr/recherche/equipes/oasis.en.html

[32] The French National Institute for Research in Computer Science and Control (INRIA). [Online]. Available: http://www.inria.fr/index.en.html

[33] Free Software Foundation, Inc. GNU Lesser General Public License. [Online]. Available: http://www.gnu.org/licenses/lgpl.html

[34] Central Institute of Applied Mathematics, Forschungszentrum Juelich, Germany. Uniform Interface to Computing Resources (UNICORE). [Online]. Available: http://unicore.sourceforge.net/

[35] The NorduGrid Collaboration. NorduGrid. [Online]. Available: http://www.nordugrid.org/

[36] OASIS Research Team, *ProActive Reference Booklet*, v3.0 ed., November 2005. [Online]. Available: http://www-sop.inria.fr/oasis/proactive/doc/release-doc/html/ReferenceCard.html

[37] G. Atkinson and G. Wells, "Benchmarking the Java-Based ProActive Grid Achitecture with a Java Implementation of the Linda Coordination Language," 2006. [Online]. Available: http://research.ict.ru.ac.za/g03a0381/Research%20Library/Grid%20Computing/gc_Atkinson_Wells.pdf

[38] A. Wollrath, R. Riggs, and J. Waldo, "A Distributed Object Model for the Java System," in *USENIX 1996: Proceedings of the 1996 USENIX Conference on Object-Oriented Technologies*, 1996. [Online]. Available: http://pdos.csail.mit.edu/6.824/papers/rmi96.pdf

[39] G. Atkinson and G. Wells, "An Investigation of Grid Computing, A Literature Review," 2006. [Online]. Available: http://research.ict.ru.ac.za/g03a0381/ResearchLibrary/GridComputing/LiteratureReview.pdf

[40] E. Gamma, R. Helm, R. Johnson, and J. Vissides, *Design Patterns - Elements of Reusable Object-Orientated Software*. Professional Computing Series. AW, 1995.

[41] Tanuki Software. Java Service Wrapper). [Online]. Available: http://wrapper.tanukisoftware.org/

[42] T. Akhurst, "The Role of Parallel Computing in Bioinformatics," Master's thesis, Rhodes University, 2004.

[43] Francis C. Moon, *Chaotic and Fractal Dynamics: An Introduction for Applied Scientists and Engineers*. John Wiley & Sons, Inc., A Wiley-Interscience publication.

[44] G. von Laszewski, I. T. Foster, and J. Gawor, "CoG Kits: A Bridge Between Commodity Distributed Computing and High-Performance Grids," in *Java Grande*, 2000, pp. 97–106. [Online]. Available: citeseer.ist.psu.edu/vonlaszewski00cog.html

# Appendix A

# Framework

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.io.Serializable;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Vector;
import org.objectweb.proactive.ActiveObjectCreationException;
import org.objectweb.proactive.ProActive;
import org.objectweb.proactive.core.node.Node;
import org.objectweb.proactive.core.node.NodeException;
import org.objectweb.proactive.p2p.service.P2PService;
import org.objectweb.proactive.core.ProActiveException;
import org.objectweb.proactive.p2p.service.StartP2PService;
import org.objectweb.proactive.p2p.service.node.P2PNodeLookup;
import org.objectweb.proactive.core.config.ProActiveConfiguration;
/**
  * @author Greg Atkinson, 2006
  */
public abstract class Controller implements Serializable {
        static final long serialVersionUID=1;

        // ** GLOBAL PARAMETERS MODIFYABLE BY THE CHILD ** //

        protected boolean logToSTDOUT=true;
        protected boolean doNoWork=false;                    // Just find and de-
stroy the nodes, no processing (for debugging)
       protected long nodeLookupTimeout=30000;        // Time in milliseconds to look for nodes
        protected long startTime,duration;
        protected StringBuffer log=new StringBuffer();
        protected boolean allParametersSent=false;
        // ** JOIN THE P2P NETWORK, GET NODE URLs & START THE DRONES ** //

        private String appID,p2pNodeFile,droneClassName;
        private int numNodesRequired=-1;

        private String[] droneURLs;
        private int numDrones=0,awaitedResults=0;
        protected Drone[] drone;
        private boolean droneActive[];
        protected Object[] droneResults;
```

```
        // Empty, no-arg constructor required by ProActive
        public Controller() {}

        public Controller(String _appID, int _numNodesRequired, String _p2pNodeFile) {
            // Process the Grid application parameters
            if (_appID==null) {die("Grid application name not set, please define 'String appID'");}
            if (_numNodesRequired<0) {die("Number of nodes required not set, please de-
fine 'int numNodesRequired'");}
            if (_p2pNodeFile==null) {die("Node list filename not set, please de-
fine 'String p2pNodeFile'");}
            appID=_appID;
            numNodesRequired=_numNodesRequired;
            p2pNodeFile=_p2pNodeFile;

            new Thread(new Runnable() {
                public void run() {

                    // Compose the class names and tell us who we are
                    droneClassName=appID+"Drone";
                    System.out.println("ProActive Grid Application : "+appID+"\n");

                    // Load the default ProActive configuration
                    ProActiveConfiguration.load();

                    // ** Connect to the P2P network and acquire nodes from it. ** //
                    // Start P2P Service
                    StartP2PService startP2PService=new StartP2PService(p2pNodeFile);
                    try {startP2PService.start();}
                    catch (ProActiveException e) {
                        logError("ProActiveException starting P2P service");
                        e.printStackTrace();
                    }
                    P2PService p2pService=startP2PService.getP2PService();
                     // Acquire Nodes
                    P2PNodeLookup p2pNodeLookup;
                    if (numNodesRe-
quired>0) p2pNodeLookup=p2pService.getNodes(numNodesRequired,appID,appID);
                    else {p2pNodeLookup=p2pService.getMaximunNodes(appID,appID);}

                    // The nodes *must* be destroyed, therefore try not let any excep-
tion halt execution
                    try {
                        logMessage("Waiting no more than "+nodeLookupTime-
out+"ms for "+numNodesRequired+" nodes.");
                        Vector nodes=p2pNodeLookup.getNodes(nodeLookupTimeout);
                        long timeWaited=0;
                        while (!p2pNodeLookup.allArrived()) {
                            try {Thread.sleep(1000);} catch (Exception e) {}
                            timeWaited+=1000;
                            if (timeWaited>=nodeLookupTimeout) {
                                doNoWork=true;
                                logError("Insufficient nodes found.");
                                break;
                            }
                        }
                         // Get the node URLs
                        logMessage("Getting node URLs...");
                        droneURLs=new String[nodes.size()];
```

```
                    for (int i=0;i<droneURLs.length;i++)
                      droneURLs[i]=((Node)(nodes.get(i))).getNodeInformation().getURL();

                    // Start timing by getting the current time
                    startTime=System.currentTimeMillis();

                    // ** Fire up the application ** //
                    if (!doNoWork)
                   if (droneURLs.length>=numNodesRequired && droneURLs.length>=1) {
                            numDrones=droneURLs.length;
                            drone=new Drone[numDrones];
                            droneResults=new Object[numDrones];
                            // Create Drones
                            if (createDrones()) logMessage("All Drones created.");
                            else logError("Failed to initialise some Drones.");
                             // Hand out processing to Drones & receive the results
                            if (control()) logMessage("All results received.");
                            else logError("Some results missing.");
                        } else logError("Insufficient nodes, shutting down.");

                         // Calculate the duration of execution
                        duration=System.currentTimeMillis()-startTime;
                    } catch (Exception e) {
                        e.printStackTrace();
                    }

                    // ** Destroy and replace the used nodes. ** //
                    // Kill used nodes (this process automatically cre-
ates a new node in the old one's place)
                    p2pNodeLookup.killAllNodes();
                     // Pause while we make sure the process completed.
                    logMessage("Waiting for the deletion of old and cre-
ation of new shared nodes...");
                    try {Thread.sleep(30000);} catch (Exception e) {}
                    logMessage("Bye.");

                     // Report the duration of execution
                    logMessage("Time of execution: "+millisecondsToString(duration));

                     // Produce the log file
                    try {
                        Date now=new Date(System.currentTimeMillis());
                        SimpleDateFormat formatter=new SimpleDateFormat("yyyyMMdd-
HH'h'mm");
                        String extension=formatter.format(now);
                        PrintWriter out=new PrintWriter(new BufferedWriter(
                          new FileWriter(appID+"."+numNodesRequired+"Drones."+extension)));
                        out.write(log.toString());
                        out.close();
                    } catch (IOException e) {
                        logError("IOException when attempting to write log file.");
                        e.printStackTrace();
                    }

                    System.exit(0);

                }
            }).start();
```

```
}

private boolean createDrones() {
      boolean returnValue=true;
       // Mark all the Drones as available
      droneActive=new boolean[numDrones];
      for (int i=0;i<numDrones;i++) droneActive[i]=false;
       // Create, initialise & start drones processing
      for (int i=0;i<numDrones;i++) {
            // Create drone
            try {
                  drone[i]=(Drone)ProActive.newActive(droneClassName,null,droneURLs[i]);
                  logMessage("Created drone on"+droneURLs[i]);
            } catch(NodeException e) {
                  logError("NodeException creating drone "+(i+1)+" on "+droneURLs[i]);
                  e.printStackTrace();
                  returnValue=false;
            } catch(ActiveObjectCreationException e) {
                logError("ActiveObjectException creating drone "+(i+1)+" on "+droneURLs[i]);
                  e.printStackTrace();
                  returnValue=false;
            }
      }
      return returnValue;
}

private boolean control() {
      boolean returnValue=true,startDrones;
      Vector<Serializable> params;
      // Do the controlling
      while (!allParametersSent||awaitedResults>0) {
            // Start the Drones if not all the parameters have been sent yet
            startDrones=true;
            while (startDrones&&!allParametersSent) {
                  // End the loop if no Drone is started
                  startDrones=false;
                  // Get parameters for a Drone
                  params=generateDroneParameters();
                  // Check we have parameters and that there is an available Drone
                  if (params!=null&&!params.isEmpty())
                        for (int i=0;i<numDrones;i++)
                              if (!droneActive[i]) {
                                    // Start the available Drone processing & make record of it
                                    droneActive[i]=true;
                                    awaitedResults++;
                                   logMessage("Drone "+(i+1)+" out of "+numDrones+" started.");
                                    droneResults[i]=drone[i].doProcessing(params);
                                    if (!ProActive.isAwaited(droneResults[i]))
                                          logMessage("Results object is not an awaited future.");
                                    // Attempt to start another Drone by continuing the loop
                                    startDrones=true;
                                    // No need to search through the rest of the Drones
                                    break;
                              }
            }

            // Check if there are results waiting
            for (int i=0;i<numDrones;i++)
                  if (!ProActive.isAwaited(droneResults[i])) {
```

```
                        if (droneActive[i]) {
                            // Pass the result object on
                        logMessage("Drone "+(i+1)+" out of "+numDrones+" returned results.");
                            receiveDroneResults((Serializable)droneResults[i]);
                            // Mark the Drone available
                            awaitedResults--;
                            droneActive[i]=false;
                        }
                    }

                // Let's not have any busy waits now...
                try {Thread.sleep(500);} catch (Exception e) {}
            }

            return returnValue;
        }

        // ** ABSTRACT METHODS WHICH ARE APPLICATION SPECIFIC ** //

        abstract protected Vector<Serializable> generateDroneParameters();
        abstract protected void receiveDroneResults(Serializable result);

        // ** MISCELLANEOUS FUNCTIONS ** //

        public void logMessage(String msg) {
            if (logToSTDOUT) System.out.println("MESSAGE: "+msg);
            log.append(msg+"\n");
        }

        public void logError(String err) {
            if (logToSTDOUT) System.out.println("ERROR: "+err);
            log.append(err+"\n");
        }

        private void die(String err) {
            System.err.println("ERROR: "+err);
            System.exit(1);
        }

        public String millisecondsToString(long milliseconds) {
            long milli,sec,min,hr;
            milli=milliseconds%1000;
            sec=milliseconds/1000;
            min=sec/60;
            sec%=60;
            hr=min/60;
            min%=60;
            return ((hr<10)?"0"+hr:hr)+":"+
                ((min<10)?"0"+min:min)+":"+
                ((sec<10)?"0"+sec:sec)+"."+
                ((milli<100)?"0":"")+((milli<10)?"0"+milli:milli)+
                " ["+milliseconds+"ms]";
        }
    }
```

---

```
    import java.io.Serializable;
    import java.util.Vector;
    /**
     * @author Greg Atkinson, 2006
```

```
 */
public abstract class Drone implements Serializable {
	static final long serialVersionUID=1;

	 // ProActive compulsory empty no-args constructor
	public Drone() {}

	abstract public Serializable doProcessing(Vector<Serializable> params);
}
```

# Appendix B

# GridMandelbrotset

```
import java.io.Serializable;
import java.util.concurrent.ConcurrentLinkedQueue;
import java.util.Vector;
/**
  * @author Greg Atkinson, 2006
  */
public class MandelbrotSetController extends Controller implements Serializable {
        static final long serialVersionUID=1;
        // ProActive empty, no-arg constructor
        public MandelbrotSetController() {}

       public MandelbrotSetController(String appID, int numNodesRequired, String p2pNodeFile) {
              super(appID, numNodesRequired, p2pNodeFile);
        }

        private MandelbrotSetUI ui;
        private ConcurrentLinkedQueue<Vector<Serializable> > jobs=new Concur-
rentLinkedQueue<Vector<Serializable> >();
        private ConcurrentLinkedQueue<MandelbrotSubset> results=new Concur-
rentLinkedQueue<MandelbrotSubset>();

        public MandelbrotSetController(String appID, int numNodesRe-
quired, String p2pNodeFile, MandelbrotSetUI ui) {
              this(appID, numNodesRequired, p2pNodeFile);
              this.ui=ui;
        }
        protected void receiveDroneResults(Serializable mandelbrotSubset) {
              results.add((MandelbrotSubset)mandelbrotSubset);
              ui.panelMandelbrot.retrieveResults();
        }

        public void addJob(Object[] p) {
              Vector<Serializable> params=new Vector<Serializable>(p.length);
              for (int i=0;i<p.length;i++) params.add((Serializable)p[i]);
              jobs.add(params);
        }

        protected Vector<Serializable> generateDroneParameters() {return jobs.poll();}
        public MandelbrotSubset getResult() {return results.poll();}
}
import java.awt.Dimension;
import java.awt.Point;
```

```
import java.net.InetAddress;
import java.util.Vector;
import java.io.Serializable;
```

---

```
/**
  * @author Greg Atkinson, 2006
  */
public class MandelbrotSetDrone extends Drone implements Serializable {
        static final long serialVersionUID=1;
        // ProActive empty, no-arg constructor
        public MandelbrotSetDrone() {}
        private MandelbrotSubset mandelbrotSubset;

        public MandelbrotSubset doProcessing(Vector<Serializable> params) {

                if (params.get(0) instanceof MandelbrotSubset) {
                        mandelbrotSubset=(MandelbrotSubset)params.get(0);
                        mandelbrotSubset.generateMandelbrotSubset();
                } else {
                        mandelbrotSubset=new MandelbrotSub-
set((Dimension)params.get(0),(Integer)params.get(1),
                                        (Dou-
ble)params.get(2),(Double)params.get(3),(Double)params.get(4),(Double)params.get(5),
                                        (Point)params.get(6),(Integer)params.get(7));
                        mandelbrotSubset.generateMandelbrotSubset();
                }
                mandelbrotSubset.setHostname(getHostname());
                return mandelbrotSubset;
        }

        public String getHostname() {
                String hostname="unknown";
                try {hostname=InetAddress.getLocalHost().getHostName();}
                catch(Exception e) {}
                return hostname;
        }
}
import java.awt.*;
import java.awt.image.BufferedImage;
import javax.swing.*;
import java.io.Serializable;
```

---

```
/**
  * @author Greg Atkinson, 2006
  */
public class MandelbrotSubset implements Serializable {
        static final long serialVersionUID=1;

        private String hostname="unknown";
        private int index;
        private ImageIcon imageIcon;
        private Dimension resolution;
        private Point position;
        private double real,imaginary,dr,di;
        private long[][] heightMap;
        private boolean heightMapGenerated=false;
        private final long maxN=10000;
        private final int numColours=200;
        private Integer colourType;
```

```
        private final Color[] spectrum=new Color[numColours];

        public MandelbrotSubset() {}

        public MandelbrotSubset(Dimension resolution, Integer colourType,
                Double real, Double imaginary, Double width, Double height,
                Point position, Integer index) {
            // Process Parameters
            this.resolution=resolution;                    // Resolution of the image to create
            this.real=real.doubleValue();                  // Top left corners of a rectan-
gle in the complex plane
            this.imaginary=imaginary.doubleValue();
            dr=width.doubleValue()/resolution.width;       // Calculate the differentials
            di=height.doubleValue()/resolution.height;
            this.position=position;                        // Where the image will be ren-
dered on the canvas
            this.index=index.intValue();
            this.colourType=colourType;
            // Generate the colour spectrum
            for (int i=0;i<numColours;i++)
                spectrum[i]=new Color(Color.HSBtoRGB((float)(numColours-1-
i)/(float)numColours,1,1));
            // Create height map
            heightMap=new long[resolution.width][resolution.height];
        }

        public void generateMandelbrotSubset() {
            if (!heightMapGenerated) generateHeightMap();
            generateImage();
        }

        private void generateHeightMap() {
            double x0=real,y0=imaginary,x=0,xx=0,y=0,maxDistance=4.0;
            long n=0;
            int i,j;
            // Generate the height map
            for (i=0;i<resolution.height;i++) {
                x0=real;
                for (j=0;j<resolution.width;j++) {
                    x=xx=y=n=0;
                    while ((n<maxN)&&((x*x)+(y*y)<maxDistance)) {
                        xx=x*x-y*y+x0;
                        y=2*x*y+y0;
                        x=xx;
                        n++;
                    }
                    heightMap[j][i]=n;
                    x0+=dr;
                }
                y0-=di;
            }
            heightMapGenerated=true;
            System.out.println("Height map generated.");
        }

        private void generateImage() {
            Image image=new BufferedIm-
age(resolution.width,resolution.height,colourType.intValue());
            Graphics graphics=image.getGraphics();
```

```
                // Map the heights to colors and draw them on the image
                long height;
                int i,j,col=0;
                for (i=0;i<resolution.height;i++) {
                        for (j=0;j<resolution.width;j++) {
                                height=heightMap[j][i];
                                if (height==maxN) graphics.setColor(Color.black);
                                else {
                                        col=(int)(height%numColours);
                                        graphics.setColor(spectrum[col]);
                                }
                                graphics.fillRect(j,i,1,1);
                        }
                }
                imageIcon=new ImageIcon(image);
                System.out.println("Image generated.");
        }
        public String toString() {return hostname;}
        public void setHostname(String hostname) {this.hostname=hostname;}
        public int getIndex() {return index;}
        public Image getImage() {return imageIcon.getImage();}
        public Point getPosition() {return position;}
}
import java.awt.*;
import java.awt.event.*;
import java.awt.image.BufferedImage;
import java.awt.image.BufferStrategy;
import javax.swing.*;
import java.util.Stack;
```

---

```
/**
  * @author Greg Atkinson, 2006
  */
public class MandelbrotSetUI {

        private final String appID="MandelbrotSet";
        private final int numDrones=9;
        private final String nodeListFile="p2pNodeFile.testbed";
        private MandelbrotSetController controller;
        protected Mandelbrot panelMandelbrot;

        public static void main(String[] args) {new MandelbrotSetUI();}
        public MandelbrotSetUI() {
                // Create Controller && Drones
                controller=new MandelbrotSetController(appID,numDrones,nodeListFile,this);
                 // Create the GUI
                javax.swing.SwingUtilities.invokeLater(new Runnable() {
                        public void run() {
                                createGUI();
                        }
                });
        }

        private void createGUI() {
                 // Construct the frame
                JFrame frame=new JFrame("Grid Mandelbrot Set");
                 // Configure things to shutdown properly
                frame.addWindowListener(new WindowAdapter() {
                        public void windowClosing(WindowEvent e) {
```

```
                    // Make a window close shut the Controller & Drones down properly
                    controller.allParametersSent=true;
                }
            });

            // Construct the tabbed pane
            JTabbedPane tabbedPane = new JTabbedPane();
            panelMandelbrot=new Mandelbrot(numDrones);
            panelMandelbrot.setSize(new Dimension(800,600));
            frame.getContentPane().add(panelMandelbrot);

            // Finalise the GUI
            frame.pack();
            frame.setVisible(true);

            // Enable double buffering & render the Mandelbrot
            panelMandelbrot.setIgnoreRepaint(true);
            panelMandelbrot.createBufferStrategy(2);
            panelMandelbrot.generateMandelbrot();
            panelMandelbrot.renderCanvas();

            // Construct the listeners
            panelMandelbrot.addMouseListener(panelMandelbrot);
            panelMandelbrot.addMouseMotionListener(panelMandelbrot);
            frame.addComponentListener(panelMandelbrot);
            tabbedPane.addFocusListener(panelMandelbrot);
        }

        public class Mandelbrot extends Canvas
                implements MouseLis-
tener,MouseMotionListener,ComponentListener,FocusListener {
            static final long serialVersionUID=1;

            private Image[] image;
            private Point[] position;
            private Graphics graphics;
            private BufferStrategy bufferStrategy;
            private MandelbrotSubset[] mandelbrot;
            private MandelbrotSubset mandelbrotSubset;
            private int numDivi-
sions,clickX,clickY,dragX,dragY,selX,selY,selW,selH,awaitedResults=0;
            private boolean busyRendering=false,busyDragging=false,resizeMandelbrot=false;
            private double real=-2.0,imaginary=1.1,realW=2.5,imaginaryH=2.2;
            private Object[] params;
            private Stack<double[]> history=new Stack<double[]>();

            public Mandelbrot(int _numDivisions) {
                numDivisions=_numDivisions;
                image=new Image[numDivisions];
                position=new Point[numDivisions];
                mandelbrot=new MandelbrotSubset[numDivisions];
            }

            public void renderCanvas() {
                bufferStrategy=getBufferStrategy();
                graphics=bufferStrategy.getDrawGraphics();
                // Clear the screen
                graphics.setColor(Color.white);
                graphics.fillRect(0,0,getWidth(),getHeight());
```

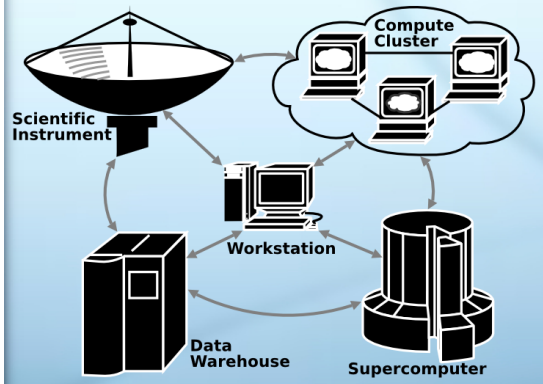
```
                // Draw the Mandelbrot
                for (int i=0;i<numDivisions;i++)
                        if (image[i]!=null)
                                graphics.drawImage(image[i],position[i].x,position[i].y,this);
                // Draw the selection box if necessary
                if (busyDragging||busyRendering) {
                        graphics.setColor(Color.white);
                        graphics.drawRect(selX,selY,selW,selH);
                }
                // Flip to the screen
                bufferStrategy.show();
        }

        public void retrieveResults() {
                int index;
                mandelbrotSubset=controller.getResult();
                if (mandelbrotSubset instanceof MandelbrotSubset) {
                        awaitedResults--;
                        index=mandelbrotSubset.getIndex();
                        mandelbrot[index]=mandelbrotSubset;
                        image[index]=mandelbrotSubset.getImage();
                        position[index]=mandelbrotSubset.getPosition();
                        if (awaitedResults==0) busyRendering=false;
                        renderCanvas();
                        if (awaitedResults==0&&resizeMandelbrot) {
                                resizeMandelbrot();
                                resizeMandelbrot=false;
                        }
                } else System.out.println("NULL Result.");
        }

        public void generateMandelbrot() {
                // Maintian the aspect ratio and without reducing the image area
                double newWidth,newHeight;
                newWidth=getWidth()*imaginaryH/getHeight();
                newHeight=getHeight()*realW/getWidth();
                if (newWidth>realW) {
                        // Enlarge width
                        real-=(newWidth-realW)/2;
                        realW=newWidth;
                } else if (newHeight>imaginaryH) {
                        // Enlarge height
                        imaginary+=(newHeight-imaginaryH)/2;
                        imaginaryH=newHeight;
                }
                // Remember zoom history
                history.push(new double[]{real,imaginary,realW,imaginaryH});
                // Distribute rendering of the Mandelbrot
                awaitedResults=numDivisions;
                for (int i=0;i<numDivisions;i++) {
                        params=new Object[8];
                        params[0]=new Dimen-
sion(getWidth(),(int)Math.ceil((double)getHeight()/(double)numDivisions));
                        params[1]=new Integer(BufferedImage.TYPE_INT_RGB);
                        params[2]=new Double(real);
                        params[3]=new Double(imaginary-
(double)i*imaginaryH/(double)numDivisions);
                        params[4]=new Double(realW);
                        params[5]=new Double(imaginaryH/(double)numDivisions);
```

```
                params[6]=new Point(0,(int)Math.floor(i*getHeight()/numDivisions));
                params[7]=new Integer(i);
                controller.addJob(params);
            }
        }

    public void mousePressed(MouseEvent e) {
            if (!busyRendering && e.getButton()==MouseEvent.BUTTON1) {
                // Obtain the mouse coordinates
                clickX=e.getX();
                clickY=e.getY();
                busyDragging=true;
            }
    }

        public void mouseDragged(MouseEvent e) {
            if (busyDragging) {
                dragX=e.getX();
                dragY=e.getY();
                // Calculate the selection box
                if (clickX>=dragX) {selX=dragX; selW=clickX-dragX;}
                else {selX=clickX; selW=dragX-clickX;}
                if (clickY>=dragY) {selY=dragY; selH=clickY-dragY;}
                else {selY=clickY; selH=dragY-clickY;}
                // Update the display
                renderCanvas();
            }
        }

        public void mouseReleased(MouseEvent e) {
            double _real1,_imaginary1,_real2,_imaginary2;
            if (!busyRendering && e.getButton()==MouseEvent.BUTTON1) {
                // Disallow mouse selections until rendering is complete
                busyRendering=true;
                busyDragging=false;
                // Obtain the mouse coordinates
                dragX=e.getX();
                dragY=e.getY();
                // Calculate the selection box
                if (clickX>=dragX) {selX=dragX; selW=clickX-dragX;}
                else {selX=clickX; selW=dragX-clickX;}
                if (clickY>=dragY) {selY=dragY; selH=clickY-dragY;}
                else {selY=clickY; selH=dragY-clickY;}
                // Calculate the new complex coordinates
                _real1=selX*realW/panelMandelbrot.getWidth()+real;
                _imaginary1=-selY*imaginaryH/panelMandelbrot.getHeight()+imaginary;
                _real2=(selX+selW)*realW/panelMandelbrot.getWidth()+real;
                _imaginary2=-
(selY+selH)*imaginaryH/panelMandelbrot.getHeight()+imaginary;
                real=_real1;
                imaginary=_imaginary1;
                realW=_real2-_real1;
                imaginaryH=_imaginary1-_imaginary2;
                // Update the display
                generateMandelbrot();
            }
        }
    public void mouseClicked(MouseEvent e) {
        double[] previous;
```

```
            if (!busyRendering && e.getButton()==MouseEvent.BUTTON3 && history.size()>1) {
                    // Revert to the previous view
                history.pop();
                previous=history.pop();
                real=previous[0];
                imaginary=previous[1];
                realW=previous[2];
                imaginaryH=previous[3];
                        // Update the display
                    generateMandelbrot();
            }
        }

        public void componentResized(ComponentEvent e) {
            if (awaitedResults==0) resizeMandelbrot();
            else resizeMandelbrot=true;
        }

        public void resizeMandelbrot() {
            image=new Image[numDivisions];
            position=new Point[numDivisions];
            generateMandelbrot();
        }
        public void componentShown(ComponentEvent e) {renderCanvas();}
        public void focusGained(FocusEvent e) {renderCanvas();}

        public void mouseMoved(MouseEvent e) {}
        public void mouseEntered(MouseEvent e) {renderCanvas();}
        public void mouseExited(MouseEvent e) {}
        public void componentHidden(ComponentEvent e) {}
        public void componentMoved(ComponentEvent e) {}
        public void focusLost(FocusEvent e) {}

    }
}
```

# An Investigation of Grid Computing

Greg Atkinson
Supervised by Prof. G. Wells
E-Mail: g03a0381@campus.ru.ac.za
WWW: http://research.ict.ru.ac.za/g03a0381/




## I. Problem Statement

Many problems in modern science require manipulation of large data sets, massive mathematical calculations or computationally intensive real-time processing.

## II. The Solution

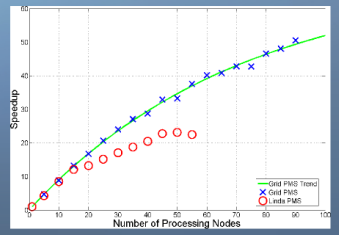
The only solution is to partition large problems and distribute them over networks of computers.

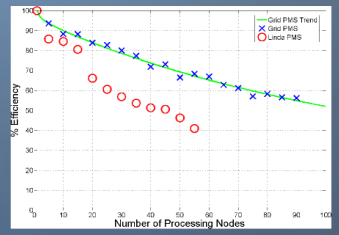
Grid computing is a network-based computing infrastructure providing
- resource access & coordination,
- access control & security,
- interfacing & monitoring,
- meta-information services,
- fault tolernace,
- storage & replica facilities,
- performance testing

and other services that enable the controlled and coordinated sharing of resources among "virtual organisations" formed dynamically by individuals and institutions with common interests.

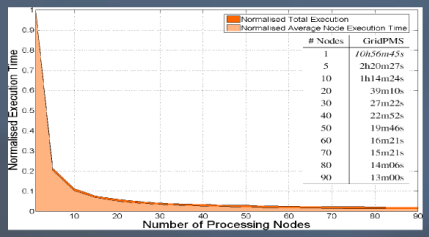
## III. Experimentation

A parallel protein motif searcher GridPMS, was constructed as the primary experiment for testing the Grid. GridPMS' task was to search the first quarter of the human genome, a 1GB data set, for specific protein sequences and DNA strings. A controlling node sliced and distributed the data set to a collection of up to 90 processing drones interconnected by a 100Mbit LAN.




The results obtained were extremely satisfying. The first chart compares the GridPMS against a similar application solving the same problem written to utilise the TSpaces framework based on the Linda coordination language. The second chart illustrates the reduction in time recorded as a greater number of Grid nodes were employed.




| # Nodes | GridPMS |
|---|---|
| 1 | *10h56m45s* |
| 5 | 2h20m27s |
| 10 | 1h14m24s |
| 20 | 39m10s |
| 30 | 27m22s |
| 40 | 22m52s |
| 50 | 19m46s |
| 60 | 16m21s |
| 70 | 15m21s |
| 80 | 14m06s |
| 90 | 13m00s |

## IV. Infrastructure

The desired infrastructure would be user friendly, portable and capable of best utilising available resources

A number of Grid infrastructures exist, the most noteworthy being ProActive and Globus. ProActive was selected due to its
- easy to use Java interface,
- simplicity to deploy,
- cluster & P2P deployment modes,
- asynchronous communication,
- remote transparent object model
- and open source policy.

ProActive
Programming, Composing, Deploying on the Grid

## V. Conclusion

The true potential of the Grid as a powerful tool for maximising the use of existing resources is finally being realised. Its unrivaled capacity for computation and resource access is providing the scientific community & commercial sector with a means to solve larger, more complex problems than ever before.

RHODES UNIVERSITY
Where leaders learn

SPONSORED BY
Telkom
Business Connexion
COMVERSE
VERSO TECHNOLOGIES
THRIP
STORTECH
tellabs