University College London
MSc Data Communications, Networks & Distributed Systems

Deployment in Computational Distributed Grids

Main Report

supervisor
Wolfgang Emmerich, Ph. D.

group members

Daisy Kong
Vesso Novov
Dimitrios Tsalikis
Stefanos Koukoulas
Thomas Karampaxoglou

06 September 2004

TABLE OF CONTENTS

TABLE OF FIGURES

ABSTRACT

Nowadays grid systems are becoming increasingly attractive as a solution for computationally intensive applications. Grid service deployment is a complex process that covers all post-development activities, such as installation, configuration and ignition of the service as well as of the infrastructure and services it depends on, de-installation, etc. However, the manual deployment is time-consuming, cumbersome and error-prone.

This project aims to investigate the use of the SmartFrog deployment framework in conjunction with the emerging Open Grid Services Architecture (OGSA) to facilitate the rapid deployment of computationally intensive applications on compute clusters. The final product will enable the automatic deployment not only of a grid application on a large collection of computational resources, but also of the infrastructure it depends on. Moreover, it will make it possible to perform various management tasks, such as re- and un-deployment, on the deployed services and infrastructure, as well as detect failures during the process and take the appropriate actions. Finally, an assessment will be made as to whether the increase, if any, in deployment speed is significant enough to offset the added time and effort needed to learn the SmartFrog technology as well as whether SmartFrog addresses every aspect (functional or non-functional) of the deployment process or further enhancements are needed.

1 INTRODUCTION

1.1 Motivation – Problem Specification

Grid computing is gaining momentum as witnessed by the tremendous amount of interest shown by academia and big software vendors. It is a services-oriented distributed computing paradigm that aims to provide massive integration and co-ordinated access to heterogeneous resources. Software components encapsulate these resources and expose them as services through interfaces. The software components, or services as called in grid computing, have to be deployed on geographically separated resources that may cross-organizational boundaries.

The issue of grid service deployment, mentioned above, is concerned with all the activities that have to be carried out after the service has been implemented and released. Such activities include the installation and ignition of the service as well as of the infrastructure and services it depends on, the appropriate configuration of the system as well as its re-configuration as part of administrative decision, de-installation, etc.

The vision of grid computing to deliver massive virtual resource pools implies the deployment of grid services on large-scale networked environments. This compounded with the fact that deployment is a complex process on its own make the grid service deployment overly costly. Some of the costs associated to software deployment include the human resources needed and the time spent to carry out all the necessary activities, the down time of the service and the disruption of the business process, the amount of computational and communication resources used.

Cutting down these costs would have been a welcomed prospect, as would a possible automation of the deployment process with the aid of a configuration and deployment framework. Such a framework would deliver a faster and perhaps more reliable process, free of component-ignition ordering errors in the light of services and components inter-dependences. It would, also, reduce the risk of misconfiguration; inconsistencies in replicated configuration files are common, as is human error. In addition, it may have provided facilities for system adaptation in cases of component failures and perhaps integrated security. One such deployment framework is SmartFrog.

1.2 Project Aims, Scope and Objectives

This project aims to investigate the use of the SmartFrog deployment framework in conjunction with the emerging Open Grid Services Architecture (OGSA) to facilitate the rapid deployment of computationally intensive applications on compute clusters. The final product will enable the automatic deployment not only of a financial grid application on a large collection of computational resources, but also of the infrastructure it depends on. Moreover, it will make it possible to perform various management tasks, such as re- and un-deployment, on the deployed services and infrastructure, as well as detect failures during the process and take the appropriate actions. Finally, an assessment will be made as to whether the increase, if any, in deployment speed is significant enough to offset the added time and effort needed to learn the Smart Frog technology.

The project's objectives are as follows:

- Familiarize with the concepts of grid computing and with the grid platform we will be using the Globus Toolkit. Deploy a grid application and gather the deployment requirements.
- Obtain both theoretical and practical knowledge about SmartFrog. Investigate what facilities it provides and how they can enable the rapid deployment of grid applications and associated infrastructure.
- Design a robust architecture that meets the functional and non-functional requirements of grid deployment. Create an implementation according to the specifications. Build test cases that aim to identify bugs in the code. Run the tests and revisit the implementation phase if required. Check the product against the specifications.

1.3 Report Structure

The following section of the report introduces Grid computing, what requirements it aims to address and what benefits it brings. The software deployment process is then detailed, as the reasons that make it a complex process. Finally, an introduction to deployment and management frameworks is given.

The tools and technologies section provides the necessary background to the reader for the better comprehension of the report. It elaborates the two basic technologies used for

the needs of our undertaking the Globus Toolkit grid platform, and the SmartFrog deployment framework.

The subsequent section details the requirements gathered at the initiation of the project, as well as its scope and objectives.

The report proceeds with project management issues. The group structure is given, and how the Unified Software Development Process (USDP) development process, which was customized to the project, helped us plan and manage the project.

The three following sections are concerned with the three phases of the project lifecycle (inception, elaboration, construction). Each section describes the activities that took place during the correspondent phase.

The next section presents an evaluation of the products of each project phase and draws conclusions, while the subsequent section highlights further work and possible enhancements to the project. Finally, the conclusion provides a summary of the development process and the project results.

2 BACKGROUND

2.1 Software Deployment Issues and Process

Traditional configuration management systems are being widely used for software development purposes. They are a means for controlling and inspecting the evolution of software systems; for tasks such as identification of a system's different components, control of the source code changes throughout the development lifecycle, audit and review [6]. However, additional mechanisms concerning software deployment activities are demanded. Software deployment is a complex process that involves all the activities that have to be carried out after a software system has been implemented and released [1]. Post-development tasks include the installation, configuration, update, adaptation, re-configuration as well as the de-installation of a system.

Software systems are no longer constructed as stand-alone applications. This is particularly the case for any scale distributed/grid system. Such systems consist of a collection of autonomous, possibly heterogeneous and geographically separated components that may have been developed by different vendors. This very architectural nature of software systems has made the deployment process significantly complicated.

The deployment of a component-based system necessitates very good knowledge of the system as a whole. Not only it requires awareness of the different components that make up the system, but also the way they interact with each other as well as the system's workflow. Consider, for example, the case in which a component may not be able to perform a function due to the fact that the component it depends on receiving a particular service has not been deployed yet. Or, an installation fails due to a missing component, e.g. absence of a decompression tool such as zip. Component interdependencies may result to the, so-called, "missing component" problem [3].

Another important deployment issue is the one of the configuration information required by software systems on a particular site [1] [3]. It should be up-to-date, accessible and able to be communicated at all times. A typical example is the adaptation of the run-time configuration of components to changes such as user profiles, shared directory structures, etc. Moreover, if the format of or access methods to the configuration information have not been standardized, makes it difficult, if not impossible, to carry out the configuration of a software system. In addition, deployment in heterogeneous platforms introduces new

challenges. The platform type should be taken into consideration as a deployment parameter. Finally, it is paramount that the correct version of components is being installed.

Resource discovery and sufficiency verification are critical aspects of the deployment process. A resource is anything (memory, disk space, cpu capacity etc) needed to enable the use of a software system at a site. It is essential that the availability and sufficiency of the infrastructure be checked against the requirements of the components making up the system. Any deployment effort will result to a partial or complete failure of the system if resource dependencies are not met.

Software deployment consists of several distinct inter-dependent tasks [1]:
Release is the task of packaging the different components of a system into a form that can be transferred to a particular site. It should also specify the resource requirements of the system for normal operation.
The installation task includes the actual transfer of the system to the consumer site and its necessary configuration before being ready for activation.
Activation is the task of running the components of a system. The activation activity might involve the activation of other systems upon which this system depends on.
De-activation is the task of shutting down the running components of the system.
Update includes the same activities as the installation task transfer and configuration of the system. It is the process of updating an older version of a system with a newer. In most cases de-activation is prerequisite.
A software system needs to be able to adapt to local environmental changes.
De-installation is the reverse of installation. It is the activity of removing the software system from a site. It prerequisites de-activation of the system.

2.2 Grid Computing

Grid computing has emerged as the latest advancement in distributed computing. It enables the sharing and aggregation of a wide range, geographically distributed resources (computational, storage, data), and presents them as a single, integrated facility for solving large-scale compute and data intensive problems [12].

The core concept behind this new distributed computing paradigm is the Virtual Organization (VO) [5] [12], which is a set of individuals and institutions under the control of sharing rules. Grid computing aims to address the requirements of such VOs [5] [8]; resource sharing in a cross-institutional setting, direct access to resources rather

than just information and file exchange, flexible control over how shared resources are used and who uses them, integrated approach for coordinated use of resources.

Grid computing offers many benefits. Vast amounts of computing power that can be used to increase dramatically the execution speeds of computationally demanding applications. Exploitation of existing under-utilized resources, such as computational capacity located at different time zones or on idle desktops. Better utilization of resources induces that less, or even no, additional investment is needed on computer infrastructure. These benefits are so significant that can offset the added complexity in developing grid applications and time needed to learn the technology [2].

It should be highlighted that the grid technology does not come as a substitute of high performance parallel computing. This is due to the high bus latency of the communication over networks and the fact that not all compute intensive problems can be partitioned in such a way that can be distributed across a collection of computers. However, grid computing finds many applications to research, such as molecular modeling for drug design, high energy physics, as well as the commercial arena, such as financial modeling and analysis in investment banking, etc [8] [15].

2.3 Deployment and Management Frameworks

The deployment issues and process presented above underline the fact that the software deployment is an overly complicated process. This gave birth to software management systems that aim to automate the process.

The deployment and management frameworks [3] [16] are able to operate in a variety of distributed and heterogeneous environments that may be under different administrative domains. Some provide means to describe and manage site resources and configuration information, which includes the hardware (storage, network etc.) resources at that particular site. Others provide mechanisms to describe, deploy and manage software systems and their configuration information, which includes system constrains and inter-dependencies between its components. Furthermore, some make it possible to monitor changes in the surrounding environment of a deployed system and take appropriate reconfiguration steps. An important point to note is that such facilities enable the management of software systems with little or no human intervention.

Due to the plethora of deployment and management systems in existence today, the choice of a particular one should be based on a set of criteria [1]. It should be identified

to which extend a deployment system covers the activities of the deployment process as well as what hooks are provided to implement unsupported or specialized ones. Coordination between deployment activities that involve distributed or composite systems, as well as notification and data exchange are key issues. The last criterion is as to whether a deployment framework can abstract over site, product and policy information. The idea is to create the deployment activities first, and then be able to carry them out to different sites, products and under different policies by just changing the associated information (i.e. the information will be given as parameters to the deployment procedures).

3 TOOLS AND TECHNOLOGIES

3.1 GlobusToolkit

The Open Grid Services Architecture (OGSA) defines a common, standard, and open architecture for developing grid applications. It aims to standardize all the fundamental conventions needed by grid applications in order to achieve transparent access to resources, portability, re-usability and interoperability. However, OGSA gives a high-level architectural view of grid services. The Open Grid Services Infrastructure (OGSI), a reference implementation of which is the Globus Toolkit, gives a more formal and technical specification.

OGSA grid technology is based on a service-oriented architecture [9]. A service is a network-enabled entity that provides some capability to its clients through the exchange of messages. It is defined by identifying the sequences of specific message exchanges that cause the service to perform its operation. Services provide access to computational and storage resources, and information through interfaces. A service-oriented architecture provides virtualization, which is the concealment of perhaps different implementations of a service behind a common interface [13]. OGSA's architecture provides access and location transparency, enables services to be accessed in a consistent way resolving any platform heterogeneity issues, and service composition.

OGSA builds on web service standards and technologies, which provide the necessary mechanisms for describing and invoking grid services [18] [8] [21]. The Web Service Definition Language (WSDL) is an XML-based description language that is used for defining web service interfaces (portTypes) []. An interface specifies the operations supported by the service and it is defined by the messages these operations consume and produce in order to provide the desired functionality. WSDL offers an abstraction over the underlying implementation of the service interface and the flexibility of defining separately transport protocol bindings and data encoding formats. The Simple Object Access Protocol (SOAP) is a lightweight XML-based protocol that can be used for the encoding and exchange of information in a decentralized, distributed environment. It is a simple enveloping mechanism that defines a framework for describing message content and how to process it, and a set of encoding rules that map user-defined data types to a transfer representation. HTTP can be used as the transport protocol, which is ideal for internet-wide distributed systems, as most firewalls will not block this kind of traffic.

However, web service standards do not address important issues related to basic service semantics, namely statefullness, discovery, dynamic service creation, lifetime management, notification and error handling [8]. All these issues are addressed by a set of standard interfaces and associated semantics specified by OGSI. These interfaces constitute the basis on which all grid services are implemented and provide service interoperability and re-usability. Thus, a grid service is essentially an OGSI-compliant web service.

OGSI [17] [10] defined interfaces is the GridSevice, which every grid service should support. It provides operations for lifetime management and access to state information of grid services. The introduction of state information is of great importance as it works around the limitation of web services that were unable to maintain state between invocations. The HandleResolver interface offers operations for obtaining references to grid services. It is essentially a naming service. The Factory interface enables the creation of transient grid service instances. Clients may decide to have solely control over the lifetime and access to the state of these services. Finally, the NotificationSource, NotificationSink and NotificationSubscription interfaces that are used for the notification of asynchronous event occurrences.

Finally, the Globus Toolkit is a platform, developed by The Globus Alliance, which can be used for the development of grid applications. It is a reference implementation of the OGSI specification. However, it also includes a lot of other facilities; system services, which are generic services used by all other grid services, a security infrastructure and various other tools, such as service browsers etc.

3.2 SmartFrog

The Smart Framework for Object Groups (SmartFrog) is a framework for describing, deploying, igniting and managing distributed systems. It consists of a description language that is used to capture system architecture and configuration information. A deployment infrastructure that implements SmartFrog descriptions and deploys software systems. A component model that defines a lifecycle and the interfaces software components should support to receive management functions. These three major aspects of the framework are elaborated below.

The SmartFrog language [16] [19] provides a means to describe a software system. A description defines the individual components that make up the system, their configuration parameters, their inter-dependences and interactions, as well as the workflow associated with the lifecycle of the components and the system as a whole. It contains an ordered collection of attributes, each having a name and a value. The value is either of primitive type (integer, string), a reference to another attribute, or a component description. The latter contains a collection of two types of attributes; template attributes that specify the component code, location and certain management aspects, and user-defined attributes that are component specific and contain its configuration information. A component description may also inherit (using the keyword extends) attributes from other components and override their attribute values. This enables the specialization, or extension with new attributes, of template descriptions for a particular context. In addition, it allows composition of multiple configuration descriptions into one. Finally, it is worthwhile noting that the language provides a flexible component linking with the aid of references.

SmartFrog's component model [16] [19] does not depend on the nature of the language described above. The interface between the framework and the notation is a parser, which implements a well-defined interface. SmartFrog considers a software system to be a, perhaps dynamic, collection of applications loosely coupled over distributed resources. Typically, applications use naming or discovery services to locate each other and must be able to deal with the case that an application they depend on has not been deployed yet. In turn, an application is a collection of components, which are tightly coupled to each other through a parent-child relationship. The parents are responsible for their children's lifecycle and are notified for their death. A component is essentially a java object that implements the Prim API, which actually constitutes SmartFrog's component model lifecycle. The framework is able to step a component through its lifecycle by invoking the API's methods (sfDeployWith( ), sfDeploy( ), sfStart( ), sfTerminateWith( )) on it. As soon as the component transitions to a different state in the lifecycle, it performs the actions specified by the implementation of the correspondent method. In an application, the lifecycle of the whole system is the combined lifecycles of the components it consists of in some sequence. The sequence depends on the parent-child hierarchy and on SmartFrog components that implement particular semantics. Such components are the Compound that defines a simple combined lifecycle, Sequence that starts the sub-components one at a time (in order to start the next, the previous should terminate), etc [22].

The SmartFrog infrastructure [16] [19] interprets software system descriptions, fed to any node participating in the SmartFrog network, and deploys components in an order, on the

nodes and with configuration parameters specified by these descriptions. Also, it steps the components through their lifecycles, while taking restorative actions should one fails. SmartFrog is flexible in to that it allows component deployment in multiple ways, without even having to change the component's code. A component is deployed by default in the root process, called SmartFrog Daemon. If the attribute sfProcessName is used in the description then the component will be deployed in a sub-process, while if sfProcessHost is included then the component will be deployed on the node indicated in the attribute's value. SmartFrog's infrastructure uses RMI as its communication base and forms a distributed network of daemons that run on each node. Apart from the deployment infrastructure, SmartFrog provides a discovery and naming service that allows components to locate each other and communicate. It, also, provides a management service via tools provided by the framework. Concluding, SmartFrog offers an environment that addresses all post-development activities, such as installing, configuring, updating, adapting, re-configuring as well as de-installing a software system.

3.3 Tomcat

The Globus Toolkit ships with a lightweight embedded and standalone-hosting environment, which provides traditional web server functionality and acts as a serlvet container. However, it is not used on production servers, in which case a standard java servlet engine, such as Tomcat is used. Tomcat [23] is bundled with the Apache Axis, which is a servlet-based implementation of the SOAP protocol that takes care of the encoding of messages send in web service requests.

3.4 Ant

Apache Ant [20] is a platform independent tool, which is used primarily for Java-based projects. It performs build tasks, such as compiling source code and in our case deploying grid services on Tomcat. It can be used instead of shell-based tools, such as Make. Ant allows the creation of tasks and rules in a XML structured file. It supports multiple targets (normally binary/executable files), dependencies (such as project 'a' depends on project 'b' so build project 'b' first), if and unless statements as decision making constructs. Users can, also, define their own build tasks by creating relevant Java classes using the native API, which makes Ant highly extensible. Ant bypasses many problems that tools such as Make have, with text-parsing ambiguity, as it uses XML for the configuration file. It can be imbedded into a variety of IDEs.

3.5 CVS

CVS [6] is a version control system. By tracking changes, it allows a complete record to be kept of a project's source file history. This history (called the Repository) is extremely useful as it is often necessary to backtrack to an older version of a source file in order to correct bugs. CVS is, also, a type of revision control. Revision control is the standard practice to maintain the progress of engineering drawings, a practice that may even be as simple as labeling an original drawing with an 'A', then a subsequent revision with a 'B', etc.

CVS is designed for projects involving many programmers. An individual programmer is able to 'check out' source files from the main repository into their own directory. At any point the programmer can choose to receive an update from the central repository, which merges any changes that may have been made into the programmer's local copy. After completing the necessary work, the programmer can 'commit' the files back to the repository.

CVS offers greater efficiency than simply backing up an entire file every time a new version is saved, a process that would require extensive amounts of storage. CVS stores all the file versions within a single file. Its efficiency lies with its ability to track and store the differing elements of all a particular file's revisions. It is then only necessary to store a single copy of the like parts shared amongst the revisions. CVS was a very helpful tool that enabled us to work together and co-ordinate our efforts.

3.6 Eclipse

Eclipse is an Integrated Development Environment (IDE) written in Java that can be used for a variety of projects and in a multitude of programming languages. It has core features that make up the base IDE and uses tools, also called plug-ins, to extend the functionality of the IDE into whatever the user desires, making Eclipse extremely flexible. Eclipse is platform-independent as it uses Java for the core IDE and the plug-ins. Its usability depends on how well the plug-ins work together, which is heavily dependant on the developer of the tools.

4. REQUIREMENTS OF THE PROJECT

Due to the nature of the format of this report, this chapter will not fully specify and describe the requirements of the project. This report follows the Unified Software Development Process, which was also used for the project, which means that in the chapters Inception phase, Elaboration phase and Construction phase there will be subsections which will focus on the specific requirements of each phase. The purpose of this chapter is to discuss and clarify the global requirements of the project. That will be done by eliciting the requirements and then by analyzing them. The conclusions that will be drawn here will be used in order to define the requirements of each iteration of USDP.

4.1 Elicitation

The process of elicitation began with very little information about the project. The reason was that the project involved many knowledge areas that were unknown to our team. So, the first step was to familiarize ourselves with those areas. The outcome of the research that was done is properly summarized in chapters 2 and 3. Suffice to say here that the goal was not to gain complete knowledge on each area but rather to have a starting point in order to decide what needed to be done in the next phase of the project.

Four basic requirements kick-started the project:

- Two technological cornerstones: From the beginning we knew that there were two specific technologies that would be involved in the project: Open Grid Services Architecture (OGSA) and SmartFrog. That was very important since it ruled out all other platforms, which supported grid services, as well as other possible deployment platforms. The focus needed to be on OGSA and SmartFrog.

- Two other related projects: A financial application which runs Monte-Carlo simulations has been implemented as a grid service by Charaka Goonatilake and may be used during the project as a testing application. A scientific application, also implemented as a grid service, by Ben Butchart may also be used during the project as a testing application. These two have been suggested to us as good candidates. They may be used and that will depend on the specific requirements that will be set for the project and after their closer inspection.

- The problem that needs to be solved: The reason that we needed to experiment with those two technologies is the inefficiency of the current deployment process of grid services. SmartFrog was introduced into the project as a way to control and hopefully amend this inefficiency. But in order to decide how we would try and solve the problem we first needed to find out where the problem lied.

- There will be no final product: The project is purely academic and there is no need to have a product as an outcome. Several prototypes may be constructed during the project but their only use would be to help us gain valuable information about SmartFrog's capabilities. The outcome of the project needs to be a set of measurements and guidelines, which will discuss whether SmartFrog can indeed help the deployment of grid services and at what degree. In other words, a comparison of deployment with SmartFrog versus the traditional process.

Using those four requirements as a starting point, interviews were conducted with the stakeholders of the project. There was a need for more information that would otherwise be unavailable to us. It is possible to learn how OGSA works by reading its documentation. On the contrary, the problems that are faced by the administrators who try and deploy grid services manually can mostly be learn through experience. The stakeholders had such experience and had faced these kinds of problems before. So we needed to know what these problems are and where they would like us to focus our attention.

During those interviews we learned that the most important problems of the traditional deployment process of grid services are the following:

- Manual deployment of infrastructure: When it comes to grid services the exact same infrastructure has to be deployed on several computers. Their number could be ten, a hundred, a thousand or even tens of thousands of computers. The process is essentially the same for each one and is simply must be repeated on each computer. This is not only annoying but also time-consuming. And time costs money. The current method of deployment is manual and that means a lot of time lost by administrators who need to sit in front of each computer and perform a complicated installation of the infrastructure. If we take into consideration a large network of computers which spans several buildings, organizations, cities, even countries, the process becomes even more time-consuming, close to impossible.

- Manual deployment of grid services: The same infrastructure can be used with all the grid services that are written with that infrastructure in mind. This means that although

it is possible to deploy one grid service or more along with the infrastructure, it is quite possible for the need to arise in the future to deploy more grid services on those same machines. This would call once again either for someone to visit each computer and install the grid service(s).

- Manual updates: The infrastructure may change or the grid service may change. The infrastructure in particular includes a variety of programs for which updates may become available after their initial installation. It would be possible to stick with the older version but not in the case of an update that fixes a serious flaw or a newly discovered security vulnerability of the program. The same goes for the grid service itself, which is essentially nothing more that a program. Newer versions may come out and that would, once again, mean another sitting in front of each computer for the administrator(s).

Having finished the interviews, it was time to complete the research and background reading in order to be able and come up with a first rough version of the requirements that follow. It should be noted that these requirements are only guidelines about the next steps of the project:

- It should be examined whether SmartFrog can be used for the deployment of grid services and their infrastructure.

- The complicated deployment of infrastructure should be automated.

- The deployment of infrastructure and grid services should be done remotely. The administrator should not have to sit in front of each computer.

- It should also be possible to update the infrastructure and grid services remotely.

- As few people should manage the network of computers that runs grid services as possible.

- The use of SmartFrog for the deployment of grid services may create issues that did not have to be addressed until now, such as security issues. Those should be identified and taken into consideration.

- SmartFrog should be used to produce a scalable system. It should be able to accommodate any number of computers.

- The possibility of automatic remote deployment should be researched inside the boundaries of a specific scenario. Such a scenario should be defined according to realistic needs of grid services administrators.

- It is not only important to examine SmartFrog's suitability, it is equally important to take into consideration the time that will be spent learning how to use SmartFrog and whether that time is worth the effort.

- The system should be platform independent. It should be possible to arbitrarily deploy them on any kind of platform and they should perform adequately in all of them.

4.2 Analysis

The outcome of the elicitation process is a set of requirements that do not even come close to the final requirements of the project. They resemble questions that needed to be answered and those answers will then be used to move the project forward. However, at this stage the answers create more questions.

- Is SmartFrog capable?

The initial studying of SmartFrog has deemed that it is capable. However, simply the fact that it is capable does not make it a better choice than the traditional deployment method. It will depend on the effort that is needed to learn SmartFrog, and the effort to write components to deploy a grid service with it. It depends on whether the process of writing those components can be automated in any way, in other words if a different component needs to be written for each grid service or whether the same component can handle different grid services.

- Can the deployment of infrastructure be automated?

It is possible, with the use of SmartFrog. Theoretically a component can handle the deployment of all the programs that are needed.

- Can it be done remotely?

Yes. And that is very important because it means that there is no need for direct access to each computer. It solves the problem of distance between the computers. The problem that is created in this case is that, if the administrator is in a remote environment he would probably not have any physical access to the computers. This means that he would be

unable to monitor the resources of each computer and handle and errors that may occur. If a computer ceases functioning the administrator would not know why.

- Are updates possible?

A grid service can be undeployed and redeployed at any time so SmartFrog can handle grid service updates. The infrastructure however will have to be uninstalled and that would ask for another component that performs that operation.

- How many administrators are needed?

There should be just one. If the deployment can happen remotely there shouldn't be any reason for more than one. However it depends on the type of control that the administrator would have.

- Can SmartFrog produce a scalable system?

Yes, it can. By automating the process and by deploying remotely the administrator may initiate deployment onto a number of computers simultaneously. Computers can be added arbitrarily at any time. However, this creates a problem of management. It would be difficult for someone to control the number of computers and keep details as to which grid services are deployed on which computers.

- Can it be platform-independent?

That would be quite difficult. The components would need to run code and scripts which are platform-dependent. However, the possibility of platform-independence should be researched.

- Other issues?

Remote deployment, especially in the case of inter-organizational deployment, security becomes an important factor. OGSA has its own security built-in and that would protect the messages, which would be exchanged as part of the actual web service. SmartFrog also provides some kind of security, which would protect the components of the system. The existence of security would, in other words, prevent an unauthorized user from taking control of the system. Both kinds of security are very important to the administrator and should be taken into consideration.

4.3 Scenario

The requirements of the project will be largely based on a scenario. However, in our case it may be better to lax the explicit boundaries of that scenario. The reason is that the conclusions that will be drawn will deal with SmartFrog in general and not for a specific use.

In that sense, regarding security it should be assumed that deployment may occur inside an organization's network in which case security is not paramount. However, it may also take place cross-organizationally and in that case security is important. Deployment may also take place over the Internet. From all these varied scenarios that call for different security requirements we choose the one, which is the toughest. Deployment over the internet. During the lifetime of this project it may be decided that such a deployment is actually not of interest to the project, possibly due to its complexity. But it should be investigated and therefore it should be considered a requirement.

Platform heterogeneity is another possible scenario that is more complicated than one which deals with a network of homogeneous operating systems. Since the USDP calls for a number of iterations, it would probably be wise to begin with an iteration that includes a homogeneous environment and then possibly explore the heterogeneous environment in further iterations. It is important though to keep it in mind because a system that manages deployment on a heterogeneous environment is better equipped to handle most networks, which contain a variety of computers running different operating systems.

Another point that should be considered is the nature of the grid services and how their use affects the resources of the computer. It is quite possible that heavy use may starve the resources and drive the operation of the computer to a halt. This is not a scenario, it is a fact. However, should the administrator, who will supposedly deploy the grid services remotely, have knowledge as to the state of resources of all the computers that he has used? The ideal answer to this, would be yes. Especially if we take into account a scenario where the computers that are used to run the grid services are also used for other reasons. In this last case it would be very important to have information regarding the resources of all the computers. That would not only allow to spot starved computers who are no longer doing any useful work but also in order to be able and get rough statistics regarding the demand for the grid services. Those statistics may actually create the need to increase or decrease the number of computers running the grid services.

This scenario does not mention many aspects of the requirements for the project. The reason is not only that we are aiming for a rather generalized approach but also because a specific scenario would not affect many requirements. For example, it is a given that deployment should occur remotely so there is no reason to consider a scenario where access to each computer would be needed. In fact, that is the problem that we are trying to solve!

4.4 Some Further Analysis

In order to get a better idea of the actions that would performed by a user of the system a Use Case diagram will be used.

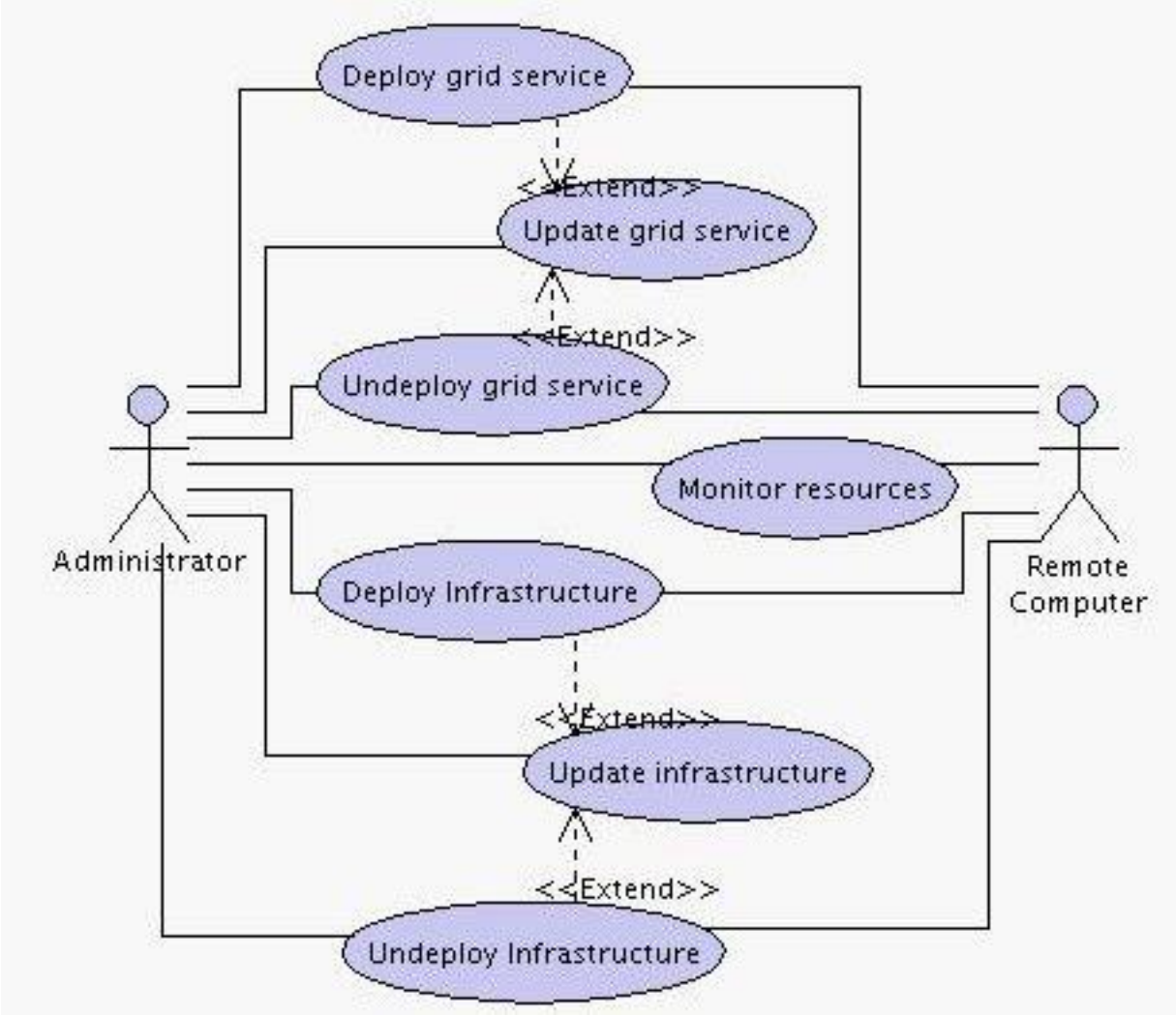

Figure 1

The “Administrator” actor is the administrator of the system. The “Remote Computer” represents any machine that the administrator may wish to control and use to deploy grid services. All the use cases have associations to both the “Administrator” and the “Remote Computer” except the “Update infrastructure” and “Update grid service” which receive functionality by extending “Deploy grid service”, “Undeploy grid service” and “Deploy infrastructure”, “Undeploy infrastructure” respectively.

This Use Case diagram gives a good idea of what the functional requirements are. They follow in greater detail.

4.5 Functional Requirements

- Deploy a grid service automatically and remotely.
- Undeploy a grid service automatically and remotely.
- Deploy the infrastructure automatically and remotely.
- Undeploy the infrastructure automatically and remotely.
- Monitor remotely the resources of a computer that runs grid services.
- Monitor remotely the state of the deployed infrastructure and grid services on each node.

4.6 Performance Requirements

- The system is scalable. It can accommodate an infinite number of computers. Computers may be added and removed at any time.
- The system is platform independent. Grid services and their infrastructure can be deployed on any computer running any operating system.

- The deployment and undeployment of grid services and infrastructure is a secure process. The administrator's control cannot be compromised at any time.

- The system can deploy secure grid services. It can build into its deployment mechanism the provision to include grid services security according to the administrators needs.

- The system is usable. The sheer number of computers does not confuse the administrator and their control is not complicated or difficult.

4.7 Design Constraints

- The system must use SmartFrog as its deployment platform.

- The system must use OGSA as its grid services implementation.

Finally, it is important to stress once again that these requirements are not meant to guide the design and development of a product. They are meant to guide the attempt to use SmartFrog in order to deploy grid services and act as measurement guidelines so that we can judge whether SmartFrog is actually a worthwhile solution. Also, this version of requirements will not be used directly for any of the following requirements subsections in each iteration. These should be considered global requirements and they are only meant as a guide for the rest of the project.

5 PROJECT MANAGEMENT

Obviously, since a group project is different from a common assignment and an individual project, the application of project management techniques was required in order to guarantee that the project would run smoothly and efficiently. The use of software engineering techniques and project management, not only ensures the careful allocation of resources, but also helps us overcome many problems and risks throughout the whole lifecycle of the project.

5.1 Group Organization and Structure

The organization chart of our group project is shown below. The project supervisor, Dr. Wolfgang Emmerich naturally becomes a key stakeholder, being active in the field of distributed system and software engineering. In addition, Paul Brebner and Ben Butchart, members of the software systems engineering group of UCL, have been working on projects related to grid computing and Globus Toolkit 3.2.

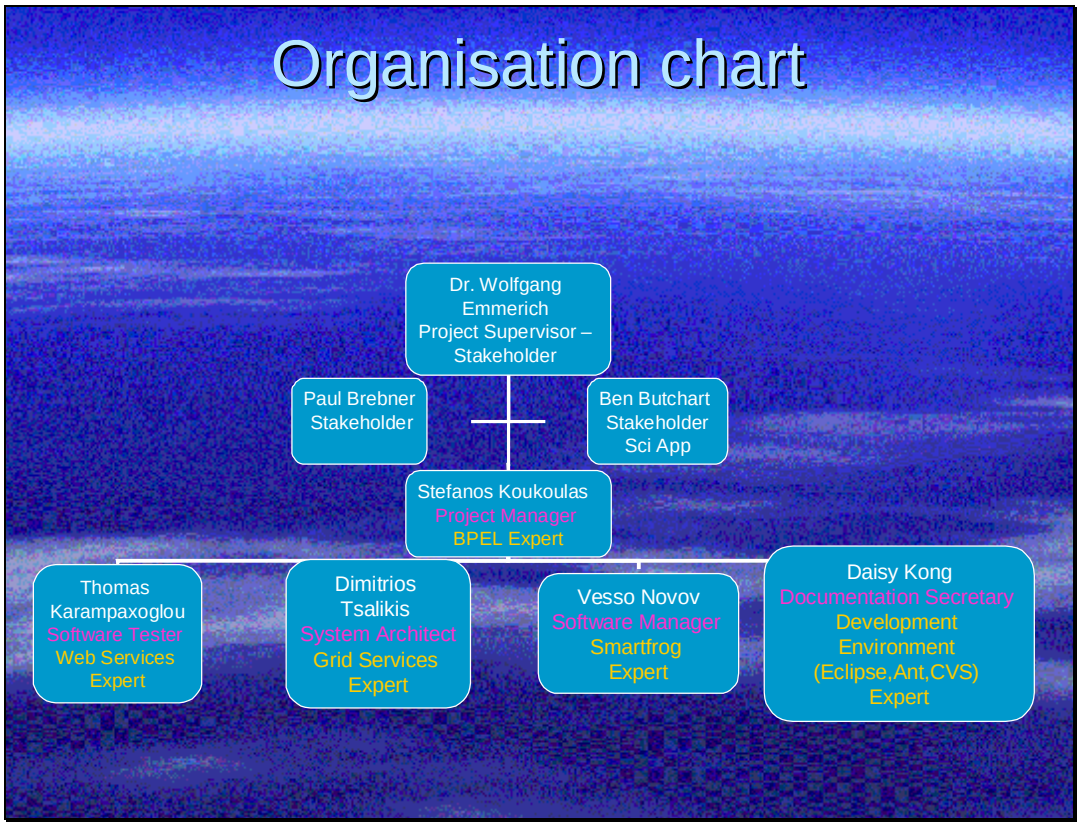

Figure 2

As far as our group's organization is concerned, every team member was assigned a different role in the management of the project. Stefanos, acting as our Project Manager, was responsible for day to day management; Thomas, the Software Tester, was responsible for building our test cases and measurements; Dimitrios, the System Architect was responsible for ensuring the system operates as a whole in terms of hardware, software and applications; Vesso, the Software Manager, was responsible for ensuring that all software platforms are correct during the whole project life cycle and finally, Daisy, was the Project Secretary, responsible for documentation.

5.2 Development Process

We chose Unified Software Development Process (USDP) as our methodology because it suited our plan to first use the simple financial application as our test case and then extend our functionality to address non- functional requirements.
The USDP lifecycle is divided into a sequence of phases. Each phase may include many iterations and each phase concludes with a major milestone. Iterations are organized into phases and contain workflows. The final result is, therefore, achieved through a sequence of iterations. Each iteration is like a mini-project covering planning, analysis and design, implementation and testing [26].

In terms of its technical aspects, USDP is an iterative and incremental process that gives considerable emphasis on the architecture of the system. Each iteration generates a release of various artifacts and deliverables that constitute the basis for the next iteration. An increment is the difference between the release of one iteration and the release of the next [26]. This Iterative approach is very controlled; iterations are planned in number, duration and objectives. We believe USDP was suitable for our project for the following reasons:

- Flexible and easily customizable
- Earlier starting risk mitigation process
- Earlier starting integration process
- Better process structure for accommodating changes
- Ensuring better final product quality

5.2.1 Flexibility

As far as flexibility is concerned, USDP is a generic software engineering process that can be customized for our project. In particular, we decided there was no point in including a transition phase, because in the beginning we didn't intend to create a final product, let alone a commercial product. The aims of the other phases are, in general, the following: in inception phase, to identify the project objectives and requirements is the main goal; during elaboration phase, we focus on the architecture of the project; finally, within construction phase, the main development and implementation of the project are carried out. Each phase included one iteration, but the elaboration phase, where we included a second smaller iteration, as well.

5.2.2 Risk Management

The iterative approach lets us mitigate risks earlier. As our project started the early iterations, it went through all process components, exercising many aspects of the project and as a result, perceived risks proved not to be risks and new risks were revealed.
Integration was not one big effort at the end of the life cycle; instead elements were integrated progressively. It was a continuous process beginning with fewer elements to integrate.

5.2.3 Accommodating Changes

The iterative approach lets us take into account requirements that change over time. It provides us with means of making tactical changes to the product – for example, we can decide to release a product early with reduced functionality. It also lets us accommodate technological changes.

The iterative approach results in more robust architecture because we keep correcting errors over several iterations. By the end, the system would have been tested several times. Flaws and performance bottlenecks would be detected in early iterations, at a time when they could still be addressed.

The development process itself could be improved and refined along the way. The assessment at the end of each iteration would not only look at the status of the project, but would also analyze what should be changed in the organization and in the process to make it perform better in the next iteration.

5.3. Project Evaluation

5.3.1 Team Communication and Status Monitoring

The group gave a considerable emphasis on the communication among the different members and on monitoring and control of the project. For each meeting of the group where important decisions were taken, the minutes were recorded. At the end of the week, a weekly group status short memo indicating the current progress and the aims for the coming week was sent out by the project manager. Then, by replying to that memo, every team member could present his own review and share different development experience with the other team members.

Furthermore, a common directory in CS file space was created, so that we could work in the same directory. There we placed any software we needed to install for the project, the minutes of the meetings, the documentation we wrote and the source code files we created. In addition, CVS would be used to ensure management of source code and concurrent access of files. At the same time, a group website was created (http://www.lukulius.com/dcnds), which had three main usages:

- track the project and the individual tasks assigned to each member
- means of communication and discussion among us
- an information centre to anyone outside the group who was interested in our project.

The website consists of three sections. The first one, general overview, contains mostly general information, which can also be viewed by anyone else who visits the website, in order to get a better idea of what our project was about. The second section, project management, has various parts that helped us get better organized. The tasks region was very important and we kept checking it regularly to see if a new task had been assigned to us or whether our current task had changed. Each member could be assigned one or more tasks with a starting and ending date. The information in the task should be sufficient so

that a meeting would not be needed to clarify it. The tasks were set by our project manager, who had the overview of the progress of the project and the individual contribution of each member. The meeting schedule contained up-to-date information about when and where the next meeting will take place and who needs to attend, since sometimes not all team members were required to attend the particular meeting, if the topic was irrelevant to their task.

The website also included an administration section, where only the members had access and could upload information. The project manager used the website to add and update tasks and to set meetings. When any team member completed a task, he logged in to the admin part of the site, set the task "completed" and wrote a short memo of what the outcome of the task was. The project manager would then review the task and may mark it as "unsuccessful", if he thought that the ideal result had not been accomplished. Tasks are classified hierarchically based on their importance. Tasks could have a high importance, which means that their successful completion was critical for the project. Tasks marked with low importance were not as critical as the highly important ones. By viewing the running tasks, our supervisor could keep an eye on our activities and what we were doing at each stage of the project. The project secretary, who was responsible for documentation, should use the admin section to upload the minutes of meetings regularly.

5.3.2 Progress Evaluation Method

The goal of measuring the progress of our project was to be able to evaluate how close or how far we were from the project's objectives in terms of completion, quality and compliance with the requirements as well as to determine how we could improve the process over time.

Considering this is only an academic research project and not a full scale commercial undertaking, the predominant way of measuring progress was through inspections and comparisons of the project milestones and percentage of work accomplished with each task identified in our group website.

In a discussion with the project supervisor we were advised against identifying and using specific metrics and collecting raw metrics data. We were advised that the Software Engineering Institute, the developer of the Capability Maturity Model, had determined that collecting and using empirical data on software metrics should be undertaken by commercial organization which project management process is evolving from CMM level 3 to CMM level 4.

At the scale of our project such activities were unsuitable and would unnecessarily overburden our efforts. Ideally, by the end of August, we expected our project management process to have matured to CMM level 2 at the most, and even that would be really difficult to achieve.

5.3.3 Quality Assessment Process

Quality Assessment Process was an integral part of our project's life cycle. This assessment process involved analytical and empirical evaluation of the degree of quality of each individual deliverable.

The way we assessed quality was by testing. Different tests had different objectives and focus. In the inception phase, we evaluated the progress through peer-views and inspections whether deliverables achieved non-functional requirements such as efficiency, scalability and robustness. At the end of elaboration phase, we evaluated how stable the architecture had become by arguing and by testing how we had addressed the non-functional requirements. In the construction phase we assessed entirely through testing how much of the functionality had been incorporated in our deliverable. The testing techniques we used will be explained in the relevant sections of chapters 7 and 8 analytically, where we will discuss our work during the elaboration and the construction phases.

5.4 Brief Description of Project Evolution

5.4.1 Time Schedule – Gantt Chart

To implement a time schedule for our project, we had to create a Gantt chart (shown below) containing unambiguous milestones. We divided the process in different phases, as we explained above. In the inception phase, there were a lot of tasks we had to have running in parallel. We managed that by splitting and distributing the work. The elaboration lasted about 4 weeks and the rest of the time was spent for the construction phase.

	Task Name	Duration	Start
1	**Inception**	**25 days?**	**Sat 24/04/04**
2	Background Reading	8 days	Sat 24/04/04
3	Prepare Presentation	8 days?	Sat 24/04/04
4	Deliver Presentation	1 day?	Thu 06/05/04
5	Conduct Interviews/Capture Project Requirements	1 day?	Tue 11/05/04
6	Create Project Requirements Doc	9 days	Tue 18/05/04
7	Prepare Project Plan/Presentation	7 days?	Tue 18/05/04
8	Install GT3.2/Run Tutorials	3 days?	Wed 19/05/04
9	**Elaboration Phase**	**28 days?**	**Fri 28/05/04**
10	Familiarize with Financial Application (FinApp)	3 days	Fri 28/05/04
11	Deploy Financial Application with GT3.2	2 days?	Wed 02/06/04
12	Comprehensive Reading - SmartFrog	6 days?	Fri 28/05/04
13	Install SmartFrog/Run Tutorials	1 day?	Mon 07/06/04
14	Feasibility Study - Financial Application/SmartFrog	1 day?	Tue 08/06/04
15	Design Prototype Architecture	4 days?	Wed 09/06/04
16	Implement Prototype Architecture	11 days?	Tue 15/06/04
17	Test Prototype Implementation	6 days?	Thu 24/06/04
18	Obtain Measurements	3 days?	Fri 02/07/04
19	**Construction Phase**	**30 days?**	**Fri 02/07/04**
20	Review/Modify Project Requirements	3 days?	Fri 02/07/04
21	Determine Prototype Enhancements	2 days?	Mon 05/07/04
22	Review/Modify Prototype Architecture	3 days?	Mon 05/07/04
23	Implement Prototype Enhacements	12 days	Fri 09/07/04
24	Test Prototype Enhancements	4 days	Tue 27/07/04
25	Test Complete System Implementation	7 days?	Wed 28/07/04
26	Obtain Measurements	5 days?	Fri 06/08/04
27	**Completion**	**17 days?**	**Thu 12/08/04**
28	Write Individual Report	3 days?	Thu 12/08/04
29	Write Final Report	15 days?	Mon 16/08/04

Figure 3

5.4.2 Risk Management Process

The risks that had impact on the project were primarily of technical and managerial nature.
The identified risks were categorized into two main groups according to the degree of control the project team had on each risk: Internal Risks, risks upon which the team had high degree of control and External Risks, risks upon which the team had little or no control.

The identified risks were classified hierarchically and assigned a value according to the probability of their occurrence: high probability of it marked as 3, medium of it marked as 2 and low of it marked as 1. The impact the risks may have on the project, had the same priority: high impact of the project marked as 3, medium of it marked as 2 and low impact of it marked as 1.

Risk control activities involved mainly two ways: one was risk avoidance – structuring the project organization in such a way as to eliminate certain risks, and the other is risk acceptance – accepting the occurrence of certain risks, but preparing either ways of mitigating their affect or contingency activities.
A Risk Register was started during the inception phase that would contain all those attributes. As scheduling and resource planning continued, the register was updated to reflect the status of the already identified as well as of newly identified risks.

6 INCEPTION PHASE

Inception Phase is the first phase of the whole project. In this phase, the main goals included establishing feasibility of the project, creating a business case, capturing key requirements, scoping the system, identifying critical risks and creating proof of concept prototype as USDP required. In our case this phase will not include a concept prototype since we do not have a final product.

As every phase was also a mini-project, due to the nature of USDP, inception includes requirements, analysis, design, implementation and test. In this phase, establishing business case, scoping the system can be considered requirements; establishing feasibility can be categorized as analysis; we did not deal with design; and test, since this is the first phase, is not applicable. [26]

From the very beginning of our project, our supervisor highlighted the fact that there are five different areas of technology conceptually involved in our project: namely Web Services, Grid Services, SmartFrog, BPEL and Development Environment (Eclipse, Ant, CVS), so each of one of us was assigned to do a background reading on a specific area. This did not mean that a member would only know about his particular subject. It was done so that we could have an initial information resource for every area of the project. The information was shared among the members of the team. This background reading activity began at the mid of April and ended early May, during this period, every team member devoted himself into the topic that he was assigned, learn as much about it as he could by searching the topic, visiting various websites, subscribing to mailing lists, reading published papers and technical manuals etc to not only know the basic concept and background of the topic but also investigate and learn the latest progress of it.

In May, a group seminar was hold, and every team member, the supervisor and another stakeholder attended this seminar. Every group member gave a presentation on the assigned topic with the use of slides in order to explain his knowledge area to the other members. At that point, the all the group members had an idea of what the project entailed. Also, our supervisor and the stakeholder were able to make sure that we have the necessary knowledge in order to continue with the next phase of the project.
Then, we had a sequence of informal and formal meetings with three stakeholders. The purpose of those meetings was to get a better idea of their interest in the project. Those meetings helped to put together the global requirements. More about that can be found in the corresponding chapter. Also, we had the opportunity to find out more about the areas of expertise and the work that they have done. The latter part was very important since

we will be dealing with their work in the next phases of the project. We were given a lot of information regarding the projects that they did and the technology that is involved in them.

Step by step, we obtained the key requirements of this project: producing an application building on Globus Toolkit 3.2 cooperating with SmartFrog to automatically deploy grid services and then to compare the solution which includes the use of SmartFrog with the traditional method which involved a lot of manual labor.

7 ELABORATION PHASE

7.1 Introduction and Considerations for the Whole Project

As we have already mentioned in our project management section, the Unified Software Development Process we followed specifies that, in general, the main goal of the elaboration phase is to create an architectural executable baseline of the system. In other words, in the elaboration phase we had to build a prototype, with which we could meet most of the functional requirements of our project. The major milestone after the end of that phase and the fundamental condition of satisfaction would be that executable architectural baseline to be considered functional, resilient and stable. Only if we were sure that our prototype and the basic functionality were tested extensively and proven stable, could we proceed to adding more functionality and enhancing the overall behavior of our system against the non-functional requirements during the construction phase.

Since we were not going to include a transition phase in our project, the main work for the building of the architecture and the development of our system would be done during the elaboration and the construction phases. We believe it is much more helpful for the reader of the report, if we follow the same sequence as our development lifecycle. For that reason, we will give a detailed presentation of what we did in the elaboration and then in the construction phase. The structure for each of the phases is the following:

- Requirements: we will establish the specific requirements for that phase. We will capture the use cases of the system and identify the main goals (in other words milestones) of the phase. Also, we will describe our basic assumptions to show what we include in our scope and what not.

- Design: We will mention the major design considerations in detail and explain how the design meets the requirements and achieves its goals. We will present different views of how we modeled the system using various UML diagrams, which are very powerful tools for conveying the type of information we want in each case.

  We will first use component diagrams to present a high level view of our architecture for each phase. Then we will display class diagrams to show how broke the required functionality into a list of classes. However, since the class

diagrams are very detailed and rather complicated, we judged it would be better to include them in our developer's section, which is part of a different document we handed in. Finally, we will use activity diagrams to describe the activities and the flows of data between activities. The activity diagrams clarify the different parallel execution threads of our system and break them out. Thus, they are particularly useful for the explanation of the Smartfrog components, because the system we created is rather complex, especially considering the fact that many of the components are Smartfrog components; thus, activity diagrams are much easier to understand than class diagrams.

§ Implementation: We will explain the process of converting the design into a real executable system. In particular, we will refer to our specific technical considerations, mainly about Smartfrog API. We will describe:

- how we implemented the design
- why we made the specific implementation choice
- what problems we encountered
- whether the choice was good or we could have done something else instead

§ Testing: We will elaborate on the testing techniques we used. In each phase we were interested in checking some specific aspects of the functionality as well as testing against the non-functional requirements. To be more specific, we chose an incremental process to implement the testing of each phase. First, we devised the test scenarios starting from simple unit tests and moving to more complex integration tests. For each scenario we will explain what we wanted to test and why. In addition, we will give examples of specific tests and the results, as it is obvious that we cannot practically fit all the test results in our report. Finally, we will evaluate if our testing process has proven that our implementation truly reflects the goals of the design, or some parts of our system are unstable and contain bugs.

Now before starting to talk about what we did in the elaboration phase according to the above structure, we will first mention some considerations and how they affected the progress of our project. At the beginning of the project, the scope specified that what we wanted to achieve was to check and prove whether we could use Smartfrog as a deployment framework to deploy grid services automatically. For that purpose, our initial plan included two grid applications we were going to use as test cases to prove our concept; the financial and the chemical application.

Since the financial application was simpler, we used it to build the executable prototype of the elaboration phase. It was exactly what we needed at that point; a basic grid application, which could allow us to create an initial baseline of our project. Therefore, the initial design we created for that phase was accompanied with a major assumption: that we were going to use the chemical application as the test case for the construction phase. So, we had to create a very generic design that could be easily extended to incorporate the specific functionality we wanted for the design of the next phase. This means that we had to keep in mind that the design we would create and the basic solution we would provide should also be usable for Ben's chemical application.

However, at some point during the implementation of the above-mentioned initial design we found out that Ben's application was quite customized in some aspects. For his own reasons, Ben wasn't using the 'standard' way of deploying a grid service. By 'standard' we mean the way that people of Globus Alliance (major developers of GT3) have established, which is the use of a grid archive (gar) file, as we have already mentioned in a previous chapter. By no means do we claim this is the only way one can deploy a grid service. Gar files are only archive files that include all the necessary files for the deployment of a grid service (GWSDL files, implementation classes, and other useful files, such as deployment descriptors). GT3 encourages the use of those files to automate the process of preparing a grid service for deployment and actually deploying it. In particular, GT3 contains Ant tasks which compile the GWSDL files and the implementation classes, generate the necessary files (WSDL files, interfaces, stubs and skeletons) and place them in specific libraries and directories, so that the Globus Toolkit runtime environment can add the grid service in the set of the deployable grid services. Also, since we chose to use tomcat as the web services container and the hosting environment, GT3 includes Ant tasks to deploy OGSA as a web service in tomcat and redeploy it, while tomcat is running, to incorporate any new grid services or changes to the existing ones without shutting the web server down.

If one chooses not to use the gar files, one has to perform all the above tasks manually. Ben, for example, decided that archiving his application files in a gar file wouldn't suit his needs, as his application involved wrapping up legacy FORTRAN code with Java classes as well as using BPEL and the Sun grid engine. He chose to devise his own customized way of archiving the files, which involved creating a jar with all the implementation classes and putting it together with all the necessary files in the same directory. From that point, he did the rest of the work manually as explained above. So, our design for the elaboration phase was oriented towards those considerations. We

included Smartfrog components to automate the manual procedure and description files to deploy those components and provide them with the suitable parameters.

As we have already said, we soon found out that having a design that can be extended to provide a generic solution for every kind of grid application, no matter how its files are archived, had no benefits for us. There was no need to reinvent the wheel and try to do with Java components what was already done with Ant scripts in GT3 core package. Therefore, in agreement with our supervisor, we decided to proceed with the definite assumption that a grid application is already compiled and archived as a gar file. At that point, we rejected using the chemical application as our second and final test case to check whether and how Smartfrog is useful in enabling automatic deployment of grid services.

The greatest benefit of that choice is that we didn't have to pay any trade-off. We didn't have to limit the requirements or the scope of the project. On the contrary, we could extend the scope and the objectives, since we would have the necessary time budget in the construction phase. The only changes had to do with our time plan. Now we had the time to learn more things about how Smartfrog works and how it can be used to achieve what we wanted. Experimenting with Smartfrog framework would allow us to create a more complete solution in terms of management of the grid services to be deployed.

We chose to explain those considerations to the reader before going into detail for each phase, so that the rest of the report makes sense and becomes self-explanatory. Now let's describe the various stages of the elaboration phase.

7.2 Requirements

7.2.1 Main Goals of the Phase

As we have already said, our objective during the elaboration phase was to create an executable prototype of our system using the financial application as test case. The prototype should, at least, incorporate the basic functionality defined for the global requirements of our project as discussed in chapter 4. Thus, and having the recently described considerations in mind, the main goal/milestone of the elaboration could be expressed as follows: to build a system that deploys a grid service automatically with

Smartfrog, by explicitly transferring the application files into specific locations in the OGSA servlet in the web server, making appropriate changes to specific web server files and restarting the web server, so that the new deployed grid service is accessible.

Next we restrict our requirements for this phase by specifying the assumptions we would take into account.

7.2.2 Basic assumptions

- The grid application is already compiled and all the necessary server-side files (jar files with the implementation classes, WSDL schemas, wsdd files) are located in the same directory

- Tomcat is used as the web server and the container of the grid services.

- Globus Toolkit 3.2 is used as the environment for building and deploying grid services. However, as far as Tomcat is concerned, Globus Toolkit is only a servlet named OGSA.

- There is a farm of machines available for hosting the new grid service. We assume each machine is clean in terms of any prerequisite environment or tools. The only necessary infrastructure consists of Java Runtime Environment (JRE) or Java Development Kit (JDK) and Smartfrog. Smartfrog is needed to enable the automatic download of all the necessary tools dynamically.

- The machines have a UNIX-like operating system, with our main focus being on Sun OS (Solaris 9) and Linux. Although there is a Network File System (NFS) in the UNIX computer system of the CS Department, we assume that each host has a separate hard disk to make the applicability of our architecture more generic.

- For the moment, we assume the machines belong to the same domain. Therefore, there are not any security restrictions stemming from the lack of trust in communications across organizational boundaries.

7.3 Design

7.3.1 General explanation of the design

It is clear that the first assumption constitutes the starting point of our design. Since all the server-side files of the grid application have already been generated and gathered in a specific place, there are certain things that have to be done for a grid service to be deployed:

- Copy the jar files to the lib directory of the OGSA servlet in tomcat, so that they can be found as new libraries

- Parse the deployment descriptor (wsdd file) of the grid service in order to find the <service> tag.

- The wsdd file is a GT3-specific XML-based configuration file used to describe which parameters the grid service deployment environment (in our case the OGSA servlet in tomcat) will use to deploy the grid service. In particular, some of the basic configuration options defined in this file are the Grid Service Handle (GSH), which is effectively the URL of the grid service, the name of the factory of the grid service, the name of the class the provides the implementation of the grid service, the name of the factory service that will create instances of the grid service and the name of the WSDL file describing the interface of the grid service.

- When the file is parsed, the whole <service> tag must be copied to the wsdd file of the OGSA servlet, which contains information of how all the grid services will be deployed.

- Copy the grid service wsdl file into a specific location in OGSA directory in tomcat called ‘schema’. The name of the grid service wsdl file is retrieved from the <instance-schemaPath> parameter of the <service> tag.

- Stop tomcat if it is already running

- Restart tomcat

After the final step, the grid service will be up and running waiting to serve requests.

Obviously, all the above presuppose that the basic support software, which we call infrastructure, is already downloaded, installed and deployed, if necessary. However, this assumption is not among the basic assumptions we mentioned earlier. Therefore, the infrastructure has to be deployed explicitly. In 'deployed', we include the essence of the three above terms together ('downloaded, installed and deployed'). Those tasks will be implemented with Smartfrog description files, which will launch correspondent Smartfrog components (RMI objects) to download and install the infrastructure files dynamically, just before the classes that will implement the above 5 steps are themselves deployed.

7.3.2 Illustration – UML Diagrams

Next we use UML diagrams to display the system we want to build in practice. Before proceeding with explaining our design in more detail, it is now time to present an overview of a system that deploys grid services automatically using Smartfrog. Note that the view of the system is deliberately kept high, so that the reader understands how the basic communication works and how Smartfrog components can be deployed remotely using Smartfrog daemons. We do not show how the deployment is done in detail. It is only clear that the Smartfrog daemon of the central node that controls the deployment is responsible for delivering the deployment request to the local daemons, which in turn create the component that will realize the deployment. When explaining our final design in the construction phase, we will show a more detailed diagram of the architecture we chose to build.

The diagram is shown on the next page as a UML component diagram. We chose this type of diagram, because it is very helpful in conveying the type of information we want at this point. In particular, it

- shows the configuration of run time processing nodes and the components that live on them
- shows a set of nodes and their relationships
- illustrates the static deployment view of our architecture

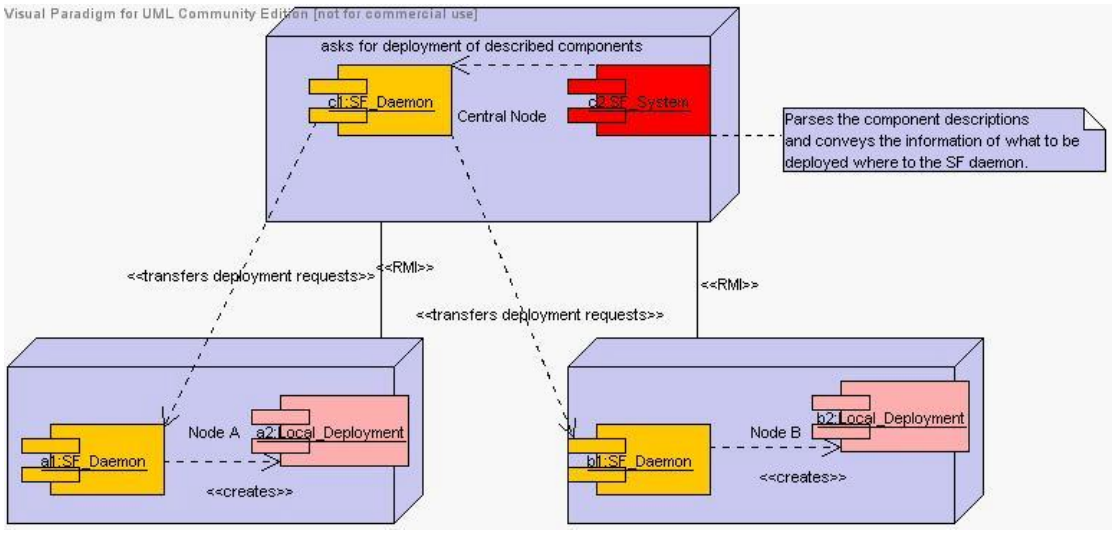

Figure 4

As we have said, the design in the elaboration phase was not extended in the construction phase, or in any way finalized to be complete and user-friendly. However, a considerable part of it would be used in the final design. Obviously, that part was the deployment of the infrastructure. Since we will explain that part when we discuss our final design in the construction phase, we will not include it in the following class diagram. As we won't explain any Smartfrog components at this stage, we don't need to include an activity diagram yet.

7.4 Implementation

Again, we will only be concerned with the explanation of the implementation issues related to the above-described classes, and not to the description files that install the infrastructure. The latter have been used and extended in the construction phase, so we will discuss their implementation process there.

First of all, we must state that the choice of the programming language we would use was pretty much predefined. Smartfrog is implemented in Java and expects that Java

implementation classes provide the functionality for its components as well. Thus, we either had to implement our functionality using another programming language and then wrap it in Java classes, or use Java from the beginning. Of course, the latter solution was much better. Java would facilitate and actually simplify the development, while the performance would not suffer at all, as we are talking about simple transformations and copying of files with no computational load.

The major challenge for this phase was the implementation of the XML parsing as well as the export and the import of XML elements from XML documents, as described in the second task of the general explanation of the design above. Researching the most effective ways to manipulate an XML document, we found out that the APIs that Sun Java provides are not really very convenient and adequate for the functionality we wanted to implement. In particular, Java includes two main XML parsers. The first is called SAX and is specified in the "javax.xml.parsers" and the "org.xml.sax" packages. The second is called Document Object Model (DOM) parser and is specified in the "org.w3c.dom" package. The advantage of the DOM parser is that it gives the chance to treat an XML element as an object of a special Java class, while the SAX parser does not provide any way of storing the info about an XML element in the memory other than as a string. In DOM, an XML document is viewed as a hierarchical structure (tree), so if we somehow extract it from a document, we can import it in a new document and insert it in a specific place or replace an older value of that tag. This is exactly what we had to do with the <service> tag, as described above. First we parsed the wsdd file of the application, exported the tag as a specific Java object, and appended it in the OGSA wsdd file as a new grid service. If we had used the SAX parser, we would have had to implement the parsing and the extraction of the value of the specific element manually, which would have involved much more work.

Apart from those two packages, there are many customized open source Java APIs for XML parsing created by developers exactly because Sun's APIs do not provide a lot of options or much flexibility. Probably it would have been even better for us to use one of those for our purpose considering the work needed to be done by our side. However, we judged that searching and learning how to use it would involve a learning curve high enough not to worth the effort and the time spent overall. Always keep in mind that we wanted to create only a prototype of our system in the elaboration phase. Therefore, it was reasonable to think that we should finish that phase as soon as possible to leave some more time for the construction phase, where the largest part of the work would have to be done.

Besides those issues, we didn't encounter any other difficulties during the implementation of the Java classes of the elaboration phase. Of course, all those classes would have to be implemented as Smartfrog components. That means that they would have to extend the basic class for Smartfrog components (PrimImpl), which, among other things, defines specific deployment lifecycle methods (sfDeploy(), sfStart() and sfTerminate()). The runtime environment of Smartfrog invokes these methods to trigger each of the stages of the deployment lifecycle of a Smartfrog component.

7.5 Testing

The testing of each module of the executable prototypes as well as the testing of the system as a whole was determined by two factors. The first was the fact we had chosen the Unified Software Development Process as our underlying project methodology. The core characteristic of this methodology is that it is iterative and incremental. Each phase of the process includes a testing artifact and each testing artifact is exercised as many times as many iterations of the phase are deemed necessary. As a result, unlike some other project methodologies, the USDP's testing process starts in the early stages of development. The second determining factor was that the system was built upon the SmartFrog framework of interlinked components, therefore it had a highly compartmentalized, object-oriented design and implementation characteristic, since SmartFrog is itself built upon object-oriented principles in practice.

Consequently from this the testing iterations started in this Elaboration phase as soon as the first fundamental building modules of this phase's prototype were developed. Those early tests verified the basic functionality expected from the components. The successful results of those tests allowed the next level of components to be created or extra complexity to be incorporated, thus, adhering to the incremental nature of the development process as a whole. Applying that same incremental methodology into the testing sessions themselves, meant that each new group of components at each new level of the component hierarchy (see developer's section) could completely rely on certain behavior and characteristic being present in the components at their base.

The executable prototype of the Elaboration phase is a system of interlinked modules each build upon groups of components delivering certain functionality. Since all of those components were SmartFrog compliant, they had to be implemented in the SmartFrog-defined language. To test the correctness of the components written in this proprietary language, a SmartFrog-provided parser was used on each individual implementation. The parsing of the implementation text files for each component guaranteed the

syntactical compliance of the code to the SmartFrog requirements as well as the compile-time correctness of the components' attribute types, attribute values and the integrity of attribute references. The successful parsing of the code of a given component guaranteed that the component (and all its sub-components) would be correctly constructed in memory as an object instance for run-time manipulation.

The parsing of components' source code was the first step of testing the system modules building blocks. The second was an actual independent, component-by-component or module-by-module execution where the visual verification of the results (whether certain files were present or not, for example) confirmed the availability of desired functionality.

The following testing scenario and results are included here as an example of the testing techniques used during the Elaboration phase. To keep this report concise only selected samples of the testing sessions and truncated test result output are shown next.

All tests were performed on the following host:

make: Dell Inspiron 5150
OS: RedHat Linux 9
CPU: 1.6GHz
RAM: 512MB
HDD: 10GB
Net: 10MB LAN
IP: 128.16.80.24
SW: SmartFrog3.02.002.beta (updated with CVS developer version as of July 12 2004)
Sun's Java(TM) 2 Runtime Environment, Standard Edition (build 1.4.2_05-b04
** An instance of SmartFrog daemon was running during all tests.*

The first action in our grid deployment process creates a directory structure on the deployment host. All downloaded and installed files/utilities/applications are placed in that working directory. The directory creation is the task of the *z15_4PrjectDirDeployer* component. As it is explained in the design section this component is built upon and it is an aggregate of SmartFrog-provided components. In particular, components that execute Unix OS commands.

The following are exerts from the configuration file *sfConfig.sf* showing the required, for this test, parameters and their values:

baseHost "128.16.80.24";

```
baseDir                 "/tmp/";
baseProjectDirName      "z15_4.2004";
baseProjectGarDirName   "gar-files";
```

The parameters defined the test would be executed on the local host "*128.16.80.24"*, under the "*/tmp/"* directory and the name of the project directory created is going to be named *"z15_4.2004"* with a subdirectory named *"gar-files"*. The first step in the test is the parsing of the source code for the component using the provided for this purpose utility *sfParse*:

```
[owner@localhost z15_4_SFDescription]$ sfParse z15_4ProjectDirDeployerTest.sf
Warning: SmartFrog security is NOT active
 Parser - SmartFrog 3.02.003_beta
Copyright 1998-2004 Hewlett-Packard Development Company, LP
    . . .
SFParse: SUCCESSFUL
[owner@localhost z15_4_SFDescription]$
```

Evidently, the parsing was successful and before the actual execution a check was made proving that the project directory did not exist at the expected place:

```
[owner@localhost z15_4_SFDescription]$ ls -lR /tmp/z15_4.2004
ls: /tmp/z15_4.2004: No such file or directory
[owner@localhost z15_4_SFDescription]$
```

Next the test was executed with the following results:

```
[owner@localhost z15_4_SFDescription]$ sfStart localhost project_dir_test
z15_4ProjectDirDeployerTest.sf
Warning: SmartFrog security is NOT active
SmartFrog 3.02.003_beta
(C) Copyright 1998-2004 Hewlett-Packard Development Company, LP
 - Successfully deployed: 'HOST localhost.localdomain:rootProcess:project_dir_test',
[z15_4ProjectDirDeployerTest.sf],  host:localhost

[owner@localhost z15_4_SFDescription]$
```

These are the truncated log messages from the above test execution (the component attempts to clear any existing directories with the same name before creating a new one):

SmartFrog ready...
[z15_4BaseInstallationUnDeployer:removeInstallationDir(bash)] LOG > Executing: rm -Rf z15_4.2004 , SFRunShell, 2
[z15_4BaseInstallationUnDeployer:removeInstallationDir(bash)] ERR > null
. . .
[z15_4ProjectDirDeployer:createDir(bash)] LOG > Executing: mkdir z15_4.2004, SFRunShell, 2
[z15_4ProjectDirDeployer:createDir(bash)] LOG > Executing: exit 0, SFRunShell, 2
[z15_4ProjectDirDeployer:createDir(bash)] ERR > null

After the execution another check for the existence of the project directory was made to verify the successful status of the test:

[owner@localhost z15_4_SFDescription]$ ls -lR /tmp/z15_4.2004
/tmp/z15_4.2004:
total 4
drwxrwxr-x 2 owner owner 4096 Aug 27 21:33 gar-files

/tmp/z15_4.2004/gar-files:
total 0
[owner@localhost z15_4_SFDescription]$

The output text above shows the successful execution of the test verifying the correct functionality of the *z15_4ProjectDirDeployer* component. During a normal deployment this action is followed by the execution of a sequence of components that download/decompress/install a number of applications/utilities which form a required base for the deployment of grid service. There is one such component for: GNU Tar, Apache Ant, Tomcat, Tomcat Deployer and OGSA. To avoid overloading this report with identical repetitive test results only the first one is documented below. The test attempted to verify that the components were capable of downloading a given resource – GNU Tar utility in that case, from a specified location – a Web server on the Internet, decompressing it and installing it under the project directory created in a manner shown on the preceding test. The following were the parameters in the configuration *sfConfig.sf* file which in addition to the ones already used with the previous test were needed:

gzTarFileName "tar-1.13.tar.gz";
gzTarUrlAddress ("http://mirrors.kernel.org/gnu/tar/" ++ gzTarFileName);
tarFileName "tar-1.13.tar";
tarDirName "tar-1.13/";

The parameters defined the exact name of the target resource *"tar-1.13.tar.gz"*, the location where it was going to be downloaded from *"http://mirrors.kernel.org/gnu/tar/*, the name and the name of the local directory where it was going to be downloaded to *"tar-1.13/"*. The source code of the component was parsed to verify the syntax:

```
[owner@localhost z15_4_SFDescription]$ sfParse z15_4GNUTarDeployerTest.sf
Warning: SmartFrog security is NOT active
Parser - SmartFrog 3.02.003_beta
(C) Copyright 1998-2004 Hewlett-Packard Development Company, LP

registerWith extends LAZY ;
urlAddress "http://128.16.80.24:8080/z15_4/tar-1.13.tar.gz";
localDir "/tmp/z15_4.2004/";
localFileName "tar-1.13.tar.gz";
localFilePath "/tmp/z15_4.2004/tar-1.13.tar.gz";
. . .
localTarFileName "tar-1.13.tar";
workingDir "/tmp/z15_4.2004/tar-1.13/";

SFParse: SUCCESSFUL
[owner@localhost z15_4_SFDescription]$
```

The output of he *sfParse* utility shows the attribute values correctly set to the parameter values in the configuration file. The actual component test was run next:

```
[owner@localhost z15_4_SFDescription]$ sfStart localhost gnu_deployer_test
z15_4GNUTarDeployerTest.sf
Warning: SmartFrog security is NOT active
SmartFrog 3.02.003_beta
(C) Copyright 1998-2004 Hewlett-Packard Development Company, LP
 - Successfully deployed: 'HOST localhost.localdomain:rootProcess:gnu_deployer_test',
[z15_4GNUTarDeployerTest.sf],  host:localhost
```

These are the truncated log messages from the above test execution:

```
[z15_4GNUTarDeployer:unTarFile:gUnZip(bash)] LOG > Executing: gunzip tar-1.13.tar.gz, SFRunShell,
. . .
[z15_4GNUTarDeployer:unTarFile:unTar(bash)] LOG > Executing: tar -xf tar-1.13.tar, SFRunShell, 2
. . .
```

128.16.9.178 (highpark.cs.ucl.ac.uk)[z15_4GNUTarDeployer:runInstall(bash)] LOG > Executing: make install, SFRunShell, 2
. . .
[z15_4GNUTarDeployer:runInstall(bash)] OUT > creating cache ./config.cache
[z15_4GNUTarDeployer:runInstall(bash)] OUT > checking host system type... i686-pc-linux-gnu
. . .
[z15_4GNUTarDeployer:runInstall(bash)] OUT > Making all in lib
[z15_4GNUTarDeployer:runInstall(bash)] OUT > make[2]: Entering directory `/tmp/z15_4.2004/tar-1.13/lib'
[z15_4GNUTarDeployer:runInstall(bash)] OUT > gcc -DHAVE_CONFIG_H -I. -I. -I.. -I.. -I. -I../intl -g -O2 -c addext.c
. . .
[z15_4GNUTarDeployer:runInstall(bash)] OUT > ranlib libtar.a
[z15_4GNUTarDeployer:runInstall(bash)] OUT > make[2]: Leaving directory `/tmp/z15_4.2004/tar-1.13/lib'

After the execution another check for the existence of the GNU Tar sub-directory in the project directory was made to verify the successful status of the test:

[owner@localhost z15_4_SFDescription]$ ls -lR /tmp/z15_4.2004/
/tmp/z15_4.2004/:
total 8
drwxrwxr-x 2 owner owner 4096 Aug 27 25:11 gar-files
drwxrwxr-x 14 owner owner 4096 Aug 25 23:22 tar-1.13

/tmp/z15_4.2004/gar-files:
total 0

/tmp/z15_4.2004/tar-1.13:
total 700
. . .
drwxrwxr-x 2 owner owner 4096 Aug 25 23:22 bin
-rw-rw-r-- 1 owner owner 50363 Jul 8 1999 ChangeLog
. . .
-r-xr-xr-x 1 owner owner 5603 Mar 2 1999 install-sh
drwxrwxr-x 2 owner owner 4096 Aug 25 23:22 intl
drwxrwxr-x 2 owner owner 4096 Aug 25 23:22 lib
-rw-rw-r-- 1 owner owner 13933 Jul 7 1999 Makefile.in
. . .
/tmp/z15_4.2004/tar-1.13/bin:
total 548
-rwxr-xr-x 1 owner owner 554394 Aug 25 23:22 tar

/tmp/z15_4.2004/tar-1.13/doc:
total 996

The executions confirmed the ability of the component to perform the core functionality of the deployment process – the ability to access and download the required resources, perform any preparatory steps before their installation and installing them at the end. These characteristics and behavior is emulated in all of the specialized component groups that comprise the executable prototype. Illustration of the tests carried for the other parts of the module, specifically the ones deploying the Apache Ant, Tomcat, Tomcat Deployer and OGSA are not included in the report as they were performed in identical manner as the test for the GNU Tar and yielded similar results. After their tests confirmed they had incorporated the expected behavior as independent components, the next step of the testing sessions focused on the prototype as a whole executable unit. The prototype itself consisted of two modules constructed by combining the lower hierarchical level autonomous components. The first module *z15_4NodeSystemBase* was a combination of the just listed components that when executed in sequence resulted in a base of installed and configured resources on which the second module *z15_4NodeSystemTier* completed the deployment process by uploading the OGSA infrastructure onto Tomcat Web Server in the form of another web application and starting that server, effectively delivering access to the OGSA -contained Grid Services to potential client requests.

The first module test was carried in the same manner as the tests before. The *sfParse* utility was used to check for syntax errors:

```
[owner@localhost z15_4_SFDescription]$ sfParse z15_4_NodeSystemBase.sf
Warning: SmartFrog security is NOT active
 Parser - SmartFrog 3.02.003_beta
(C) Copyright 1998-2004 Hewlett-Packard Development Company, LP

sfCodeBase "default";
sfClass "org.smartfrog.sfcore.workflow.combinators.Sequence";
sendTo extends LAZY ;
registerWith extends LAZY ;
actions extends LAZY  {
      installationSequence1 extends  {
            sfCodeBase "default";
            sfClass "org.smartfrog.sfcore.workflow.combinators.Sequence";
 . . .
mainGUIeventQueue LAZY HOST localhost:test_event1:gui_eq;
 SFParse: SUCCESSFUL
[owner@localhost z15_4_SFDescription]$ sfParse z15_4_NodeSystemBase.sf
```

Next the module was executed expecting a full deployment of all required utilities. The configuration parameters cold be found in the *sfConfig.sf* file:

```
gzAntFileName           "apache-ant-1.6.1-bin.tar.gz";
gzAntUrlAddress         ( "http://apache.rmplc.co.uk/dist/ant/binaries/" ++ gzAntFileName );
antDirName              "apache-ant-1.6.1/";

gzTomcatFileName        "jakarta-tomcat-5.0.25.tar.gz";
gzTomcatUrlAddress      ( "http://apache.rmplc.co.uk/dist/jakarta/tomcat-5/v5.0.25/bin/" ++
gzTomcatFileName );
tomcatDirName           "jakarta-tomcat-5.0.25/";
tomcatUsersFileName     "tomcat-users.xml";

gzTomcatDeployerFileName      "jakarta-tomcat-5.0.25-deployer.tar.gz";
gzTomcatDeployerUrlAddress    ( "http://apache.rmplc.co.uk/dist/jakarta/tomcat-5/v5.0.25/bin/" ++
gzTomcatDeployerFileName );
tomcatDeployerDirName         "jakarta-tomcat-5.0.25-deployer/";

gzOgsaFileName          "ogsa-3.2.tar.gz";
gzOgsaUrlAddress        ( "http://www-unix.globus.org/ftppub/gt3/3.2/3.2.0/gt3_core/bin/" ++
gzOgsaFileName );
ogsaDirName             ogsa-3.2/";
ogsaWebAppName          "ogsa";
```

These are truncated output messages for the completed test verifying the successful execution of all components as expected:

```
[z15_4ProjectDirDeployer:createDir(bash)] LOG > Executing: mkdir z15_4.2004, SFRunShell, 2
[z15_4ProjectDirDeployer:createDir(bash)] LOG > Executing: exit 0, SFRunShell, 2
[z15_4ProjectDirDeployer:createDir(bash)] ERR > null
. . .
[z15_4GNUTarDeployer:unTarFile:gUnZip(bash)] LOG > Executing: gunzip tar-1.13.tar.gz, SFRunShell,
[z15_4GNUTarDeployer:unTarFile:gUnZip(bash)] LOG > Executing: exit 0, SFRunShell, 2
[z15_4GNUTarDeployer:unTarFile:gUnZip(bash)] ERR > null
[z15_4GNUTarDeployer:unTarFile:unTar(bash)] LOG > Executing: tar -xf tar-1.13.tar, SFRunShell, 2
[z15_4GNUTarDeployer:unTarFile:unTar(bash)] LOG > Executing: exit 0, SFRunShell, 2
[z15_4GNUTarDeployer:unTarFile:unTar(bash)] ERR > null SFRunShell, 1
 [z15_4GNUTarDeployer:runInstall(bash)] LOG > Executing: make install, SFRunShell, 2
. . .
[z15_4BashUnTar(bash)] LOG > Executing: /tmp/z15_4.2004/tar-1.13/bin/tar -xzf apache-ant-1.6.1-
bin.tar.gz, SFRunShell, 2
[z15_4BashUnTar(bash)] LOG > Executing: exit 0, SFRunShell, 2
[z15_4BashUnTar(bash)] ERR > null
```

. . .
[z15_4BashUnTar(bash)] LOG > Executing: /tmp/z15_4.2004/tar-1.13/bin/tar -xzf jakarta-tomcat-5.0.25.tar.gz, SFRunShell, 2
[z15_4BashUnTar(bash)] LOG > Executing: exit 0, SFRunShell, 2
[z15_4BashUnTar(bash)] LOG > Executing: /tmp/z15_4.2004/tar-1.13/bin/tar -xzf ogsa-3.2.tar.gz, SFRunShell, 2
[z15_4BashUnTar(bash)] LOG > Executing: exit 0
, SFRunShell, 2[z15_4BashUnTar(bash)] LOG > Executing: /tmp/z15_4.2004/tar-1.13/bin/tar -xzf jakarta-tomcat-5.0.25-deployer.tar.gz, SFRunShell, 2
[z15_4BashUnTar(bash)] LOG > Executing: exit 0, SFRunShell, 2
[z15_4BashUnTar(bash)] ERR > null
. . .
[z15_4OGSADeployer:runInstall(bash)] LOG > Executing: source /tmp/z15_4.2004/ogsa-3.2/etc/globus-devel-env.sh, SFRunShell, 2
[z15_4OGSADeployer:runInstall(bash)] LOG > Executing: exit 0, SFRunShell, 2
[z15_4OGSADeployer:runInstall(bash)] OUT > Buildfile: build.xml
[z15_4OGSADeployer:runInstall(bash)] OUT > launchers:
. . .
[z15_4OGSADeployer:runInstall(bash)] OUT > BUILD SUCCESSFUL
[z15_4OGSADeployer:runInstall(bash)] OUT > Total time: 19 seconds
[z15_4OGSADeployer:runInstall(bash)] ERR > null

The project directory on the deployment hosts was checked to confirm all files and subdirectories were correctly downloaded and configured:

[owner@localhost z15_4_SFDescription]$ ls -l /tmp/z15_4.2004/
total 24
drwxr-xr-x 6 owner owner 4096 Feb 12 2004 apache-ant-1.6.1
drwxrwxr-x 2 owner owner 4096 Aug 25 11:52 gar-files
drwxrwxr-x 11 owner owner 4096 Aug 25 11:53 jakarta-tomcat-5.0.25
drwxrwxr-x 5 owner owner 4096 Aug 25 11:53 jakarta-tomcat-5.0.25-deployer
drwxr-xr-x 13 owner owner 4096 Aug 25 11:54 ogsa-3.2
drwxrwxr-x 14 owner owner 4096 Aug 25 11:53 tar-1.13
[owner@localhost z15_4_SFDescription]$

The results above prove the test was successful and all base applications needed for the next module *z15_4_NodeSystemTier* were present. For this test also, the file *z15_4_GridArchiveFileNames.sf*, having the parameters defining the first grid service to be deployed into OGSA infrastructure, was used:

baseGarUrlAddress "http://127.0.0.1:8080/z15_4/gt3_asian_AsianSpreadOption.gar";
baseGarFileName "gt3_asian_AsianSpreadOption.gar";
baseGarId "gt3_asian_AsianSpreadOption";

Next the last test for the prototype in this project phase was run probing the functional characteristics expected to be available before the start of the Construction phase:
The parsing:

```
[owner@localhost z15_4_SFDescription]$ sfParse z15_4_NodeSystemTier.sf
Warning: SmartFrog security is NOT active

Parser - SmartFrog 3.02.003_beta
(C) Copyright 1998-2004 Hewlett-Packard Development Company, LP

sfCodeBase "default";
sfClass "org.smartfrog.sfcore.workflow.combinators.Sequence";
sendTo extends LAZY ;
registerWith extends LAZY ;
actions extends LAZY  {
      deployGridSrvc extends  {
            sfCodeBase "default";
            sfClass "org.smartfrog.sfcore.workflow.combinators.Try";
 . . .
mainGUIeventQueue LAZY HOST localhost:test_event1:gui_eq;

SFParse: SUCCESSFUL
[owner@localhost z15_4_SFDescription]$
```

And the execution of the module itself:

```
[z15_4BaseGarDeployer:runInstall(bash)] LOG > Executing: /tmp/z15_4.2004/apache-ant-1.6.1/bin/ant
deploy -Dgar.name=/tmp/z15_4.2004/gar-files/gt3_asian_AsianSpreadOption.gar, SFRunShell, 2
[z15_4BaseGarDeployer:runInstall(bash)] LOG > Executing: exit 0, SFRunShell, 2
[z15_4BaseGarDeployer:runInstall(bash)] OUT > Buildfile: build.xml
[z15_4BaseGarDeployer:runInstall(bash)] OUT > deployGar:
. . .
[z15_4BaseGarDeployer:runInstall(bash)] OUT > BUILD SUCCESSFUL
[z15_4BaseGarDeployer:runInstall(bash)] OUT > Total time: 9 seconds
[z15_4BaseGarDeployer:runInstall(bash)] ERR > null
. . .
[z15_4WebServicesDeployer(bash)] LOG > Executing: /tmp/z15_4.2004/apache-ant-1.6.1/bin/ant
deployTomcat -Dtomcat.dir=/tmp/z15_4.2004/jakarta-tomcat-5.0.25/, SFRunShell, 2
[z15_4WebServicesDeployer(bash)] LOG > Executing: exit 0, SFRunShell, 2
[z15_4WebServicesDeployer(bash)] OUT > Buildfile: build.xml
[z15_4WebServicesDeployer(bash)] OUT > deployTomcat:
. . .
```

[z15_4WebServicesDeployer(bash)] OUT > BUILD SUCCESSFUL
[z15_4WebServicesDeployer(bash)] OUT > Total time: 4 seconds
. . .
[z15_4WebServicesDeployer(bash)] LOG > Executing: /tmp/z15_4.2004/jakarta-tomcat-.0.25/bin/startup.sh, SFRunShell, 2
[z15_4WebServicesDeployer(bash)] ERR > null

The test verified the expected functionality – the grid service file specified in the configuration file above *gt3_asian_AsianSpreadOption.gar* was successfully deployed onto OGSA infrastructure, the OGSA infrastructure was then mounted onto the Tomcat installation and the Web Server itself started. The test was the last one of the sequence of tests attempting to check whether the components the dependent on them groups of components and dependent on them module, all comprising the prototype built at the end of the Elaboration phase, had the behavior and operational characteristics as defined in the design documentation and whether the implementation followed the design specifications. The test confirmed the abilities of the prototype to deploy a grid service on a given deployment host with minimum interaction from the user.

8 CONSTRUCTION PHASE

8.1 Requirements

The goal of our design for the construction phase includes the basic goal of the design for the elaboration phase, which is essentially the primary goal of our project: to create a system that deploys a grid service automatically with Smartfrog. However, by no means do we believe this is enough. The deployment of a grid service must be accompanied by a more sophisticated functionality, which will allow the user to:

- Choose which grid service to deploy
- Choose the machines (servers) the grid service will be deployed on
- Check the resources of a server before choosing it as a candidate for hosting a grid service as well as after it is chosen. The check must be periodic to notify the user of the new situation.
- Manage the deployment of the grid service. That means check the status of deployment on a particular server and possibly proceed to action (for example, if something failed or if the resources are considered inadequate).
- Select the type of deployment of a grid service. The type of deployment depends on whether the infrastructure is already installed and tomcat is running or not.
- Undeploy a grid service or the whole infrastructure to clean the server. Manage the undeployment in the same way as the deployment.

In general, demonstrating that Smartfrog, as a general-purpose deployment framework, enables the deployment of grid services automatically was not the only objective of the project. As we will show later, Smartfrog provides that functionality. What we wanted to experiment on were the capabilities of Smartfrog in terms of the management of the deployment and other non-functional requirements, such as:

- Scalability
- Security
- Fault-tolerance
- Resource awareness
- Heterogeneity

For that reason, we wanted to build a demonstration system that encompasses the above-explained functionality. We want to emphasize the term ‘demonstration’. When setting the requirements for that phase, he had certain things in mind about the capabilities we wanted the user to have. As we will explain shortly, we decided some things should be controlled by the user and some other by the system. Those considerations are not significant at all. What is of paramount importance is to show which management capabilities Smartfrog can offer to give the developer or the user a greater level of control over what is going on.

8.1.1 Basic assumptions

The only difference with the assumptions listed for the elaboration phase concerns how the application files are packaged. As we have already explained above, we assume that the grid application has already been compiled and packaged as a gar file using straightforward Ant tasks. The rest of the assumptions (numbered 2-5) remain essentially the same in this phase as well.

We would like to expand on the issue related to the security restrictions that may occur across organizational boundaries and might prevent the actual communication from being achieved. One of our assumptions (number 6) states that all the available for deployment servers belong to the same organization. We included that to ensure that we wouldn’t have to deal with situations where we would want to test our system on machines in different domains, because a lot of problems would come up that wouldn’t have to do us. The most important of these problems would be the refusal of firewall administrators to open specific ports, so that RMI communication between Smartfrog components can take place across different domains.

Furthermore, our primary goal was to develop a system with the minimum requirement to run on UNIX machines, because that was the kind of machines we could use in the labs of the Computer Science Department. Therefore, we had to at least make sure we would not create a system that used Solaris-specific features and then could not run on Linux machines, just because the labs we worked in had Solaris machines. As far as the requirements stage is concerned, enabling the deployment of our system on a Windows environment was not assigned a high priority. We just wanted to first secure the deployment on UNIX and if we had time, we would extend the applicability of our system on a Windows environment as well. However, the most important thing was to be able in the end of the phase to tell and document which specific implementation choices

were OS-dependent and what changes or specific extensions needed to be done to make our system work under Windows.

8.2 Design

8.2.1 General explanation of the design

The basic architectural elements that the design of our system should include to meet the specified requirements are the following:

A. Components for the deployment of the infrastructure and the grid service on the remote servers
B. Components for checking the resources on the remote servers periodically and reporting the results
C. A management console that resides on a central machine and lets the user view and control the deployment of the grid service on the remote servers
D. A mechanism that enables the communication between the management console (C) and the components on the remote servers (A, B)

We now provide an explanation of the designing features of each one of these elements.

A. Deployment of the infrastructure and the grid service

Since we have decided to use Smartfrog as our deployment framework, it is evident that the components that will actually implement the deployment on the remote servers must be Smartfrog components. That means that we either have to create our own classes with the functionality that we want them to incorporate, or we can use pre-existing components that come with the Smartfrog installation package.

Smartfrog includes several APIs that help in specific areas of deployment, as well as descriptions of their classes, so that they can be used in customized description files via the use of prototypes and parameterization. There is no point in explaining how this mechanism works. We just need to say that it resembles the inheritance in Object Oriented Programming. Hence, we can use the Smartfrog descriptions as the base ‘type’ (‘prototype’ in Smartfrog terminology) of our component descriptions and then override

some of the attributes or add some more in order to achieve the desired behavior. For detailed explanation about the Smartfrog component descriptions and the parameterization mechanism the reader is requested to refer to Smartfrog Notation in Smartfrog reference manual (Hewlett-Packard Development Company 7 Jul 2004, p 12-32).

First of all, the fundamental designing consideration we had to make was related to the general functionality those components should accomplish. What options did we want the management console to have when requesting remote deployment? Would we include the deployment of the infrastructure and the grid service in the same description file or would they have to be invoked separately? What about the undeployment?

We had to answer several questions like these to come up with the design of the remote components. We decided that the management console should be able to invoke:

- Full deployment: first deploy the infrastructure and then the grid service. The deployment of the infrastructure is treated as an atomic component, which means that it either terminates successfully or all the changes are undone and the machine is left clean, as it was at the beginning. If it is successful, the grid service can then be deployed as an atomic component again.

- Partial deployment: deploy only the grid service, assuming that the infrastructure is already installed, so it's not needed to be reinstalled. If Tomcat is already running, do not stop it and restart it. Instead, deploy the grid service without affecting tomcat.

- Full undeployment: undeploy the whole infrastructure and leave the machine clean.

- Partial undeployment: Undeploy the specific grid service only, without affecting tomcat or the whole infrastructure.

By deploying the infrastructure, we mean the following sequence of actions:

- Download, untar and install GNU Tar, the implementation of the Tar program, which is used for manipulating tar archives. Usually, most of the UNIX machines include tar as an internal command. However, the installed Solaris operating system on the machines we used in the lab did not include a fully featured version of the Tar program. In particular, we tried to untar files we

downloaded from the internet, but we were getting strange errors, until we found out that the default-installed version of the Tar program had bugs. Therefore, we had to use GNU Tar to be able to untar all the rest of the infrastructure.

- Download, untar and install Apache Ant

- Download, untar and install Apache Tomcat, which will be used as our grid service container

- Download, untar and install Globus Toolkit. After that, deploy Globus Toolkit as a servlet in Tomcat, using an Ant target.

Having done all that, we say that the infrastructure is deployed. The next logical action is to deploy a grid service. To do that, we need to first call an Ant target to deploy the grid service in Globus Toolkit and then another Ant target to redeploy OGSA, which is the Globus Toolkit seen as a servlet in the web server, in Tomcat. The full undeployment requires the deletion of all the installation files of the infrastructure, while for the partial deployment we only need to first call an Ant target to undeploy the grid service in Globus Toolkit and then another Ant target to redeploy OGSA in Tomcat.

The actual way we will accomplish all those design decisions will be presented in following subsections.

B. Checking the resources

First of all, we need to explain why we want to provide this element in our system. When a pool of machines is available to host grid service, not all of the machines may meet some minimum requirements that will allow them to serve the grid service requests. At some point, a machine may be overloaded, so its response time for a grid service invocation can be unacceptable. In the same way, a machine may run out of memory, which will again result in a bad service provision or no service provision at all. Thus, we want to track these situations and act, if we think it is necessary. An inspiring example is the load balancing technique, which is very commonly used in distributed systems nowadays. If, at some point, a server is overloaded, a load balancer comes into play and redistributes the service requests to other machines.

There are also cases where we want to have info about the status of the deployment on a remote server. For example, we would like to know if the infrastructure is installed and if OGSA is running in order to trigger the appropriate type of deployment on a remote server. In the last case we do not exactly check any resources, but we choose to call the whole of this functionality as ‘resources check’.

For all these reasons we included this functionality in our design. Again, since we want those components to be executed remotely but triggered from a central machine, they had to be designed as Smartfrog components. The ‘resources’ or the ‘status’ we want to check on each server consist of the following:

- Available memory on the machine. Because our remote components will actually be Java programs, in practice we are interested in the available free memory within the Java Virtual Machine (JVM) and not the available RAM in the operating system.

- CPU load. As we said above, we want to know how loaded each server is.

- Free hard disk space. It plays an important role mainly when we consider deploying the infrastructure on a server, because the infrastructure needs quite a lot of hard disk space.

- Whether the infrastructure is installed. This would help us a lot when we have to decide if the deployment needs to be full or partial to avoid reinstalling the whole infrastructure.

- Whether OGSA is running or not. If it is, we know that Tomcat is running (as it hosts OGSA) and the grid services are also up and running.

We will not cover how we achieved to retrieve all this information from a remote server, as this is an implementation issue. However, what we need to consider in the designing stage is what level of detail the information that the remote components return to the central server will have. In particular, we need to answer whether it is useful if the remote component just sends back the specific number of a particular check to the management console or only a general feedback indicating whether the result was satisfactory or not. For example, if the available memory is 20MB, should the remote component return that number or just a simple string showing that the memory is enough?

Clearly, each option has advantages and disadvantages. Returning the exact number allows a higher level of control. As we will show shortly, the user can see this number and have a clearer image about the resources in the remote machine. On the other hand, we must take into account the fact that at some point the number of servers may be large. If the user sees a panel with a continuous flow of numbers, would not be very helpful but rather useless. He would have to constantly check the numbers on the panel to be able to act, which is not very convenient or practical. For that reason we decided to send only an indication whether the particular resource was enough or not. The decision for that would be made by the remote component directly.

C. Management Console

The management console contains a Graphical User Interface (GUI) to allow interaction with the user as well as an underlying functionality that supports this interaction and the central management of the deployment. From now on, we will use the terms ‘management console’ and ‘GUI’ interchangeably. Of course, there must be no confusion between our ‘management console’ and the ‘Smartfrog management console’, which is something completely different. We will explain the Smartfrog management console later. In general, we will use this term to refer to our management console.

In the same way as before, we want to explain the reasons for adding the functionality of a management console to our system before discussing any designing considerations. The main goal is to have a central point of the management of the deployment. As we have stated several times, the basic objective of our projective was to implement the automatic deployment of a grid service on many servers using Smartfrog. The automatic deployment, however, has no point, if you have to go to each server and start the description files, as we did in the prototype we created in the elaboration phase. This is the first step towards automatic deployment, but not the final. It is absolutely essential to be able to start and control the deployment from a central point. This capability becomes additionally important in cases where the servers are not in the same location, bur rather in distant premises from each other; they may even belong to different organizations.

Moreover, apart from a central point of control, we want to have the chance to manage the deployment. We would like the user to be able to see how the deployment progresses and act, if he thinks it is necessary. This is a key element of our system, since everything changes so quickly during the deployment on a large number of machines. We may start with a number of servers and, at some point, decide we want to remove a server or add another one. Or we may wish to deploy two grid services on some servers and one on

some others. In general, the conditions can change rapidly during run-time. Therefore, we don't only want to deploy something automatically in the least possible time. We need to be able to manage the deployment using dynamic information as our compass.
The final reason for creating the management console was because we want to let the user/administrator control a number of factors. It would be easier for us to create a central component, which would manage the whole process, but would take all the decisions transparently. What is really vital, is to let the user decide what needs to be done and not to hardcode the decisions in all possible cases in our system. The GUI must be user-friendly and in the same time functional to give all the necessary information to the user and the chance to configure the deployment the way he wants at run time.

Functionality of the management console

Next we give an overview of the functionality we want the management console to incorporate. We won't explain how the GUI works in detail, because we are interested in describing our designing decisions and considerations at this stage. From the User Guide the reader can have a clearer demonstration and understanding of the GUI.

At first, the GUI shows a list of all the available servers for deployment. The user selects some of them to be added to the GUI. The GUI sends a request to start the remote component that checks the resources on each of these servers automatically. The user can see the results of the resources check and chooses one type of deployment depending on these results or other initial information about the state of the servers (we will explain how this information is obtained shortly). This means that if the infrastructure has been installed, the user will choose partial deployment for that server.

When the deployment starts, the remote deployment components send messages continuously notifying the management console whether the specific step of the deployment process was successful or not. These messages are displayed to the user. There is an output window that shows the entire flow of messages to the user, and another window that shows only the messages that constitute warnings or errors. This way the user can view all the messages about errors in deployment or low level of resources in one place and can understand far more quickly and more securely whether he needs to act somehow.
The GUI must allow the user to view more details about a server. In particular, it is important to store the profile of server, which contains information about:

- the resources – if the level of each resource is above the minimum or not
- if the infrastructure is installed or not

- if OGSA (and consequently Tomcat) is running or not
- which grid services have been deployed on the server

Obviously, to be able to obtain this information, the management console must process all the incoming messages and update the specific server profile dynamically.

We chose not to give the GUI the ability to send a message to a remote component. Only the remote components send back feedback to the GUI about the resources of the servers or the state of the deployment. Similarly, we judged there is no point in allowing the GUI to 'terminate' a remote component (using the terminology of Smartfrog). The reason for this choice is very simple. In order to start the program, the user has to start the Smartfrog daemon, which encompasses a window and allow him to start a Smartfrog management console for any host. The latter enables the user to view the status of any component on any host with details, as well as terminate it, if he wants. Consequently, we did not need to reinvent the wheel and provide the same functionality as Smartfrog. Instead, we wanted our program to be complementary to Smartfrog. We aimed at creating a management console to have a central point of gathering of the information about all the servers. The user can utilize it to see what is going well and what not. If he wishes to retrieve more details about a remote component (for example, why it failed during deployment), he can use the Smartfrog management console to view the status of that particular host and proceed with more drastic action, such as termination.

Furthermore, another characteristic we would like the GUI to have is to store all the messages in a log file. As the flow of messages on the output window may be overwhelming and confusing, the user may use the log file to see what has happened so far in terms of the deployment; which servers have been used and at which stage the deployment is for each of them. Also, we want the profiles of the servers to be stored in a different file, so that the GUI retrieves this information when it starts. One could well argue that information about the resources or if OGSA is running are not useful during start-up, because a lot of things may have changed since the profiles were stored and this information is also available through the dynamic resources check. However, we considered the significance of knowing whether the infrastructure is installed when the console starts, as this would allow the user to choose partial deployment instead of full and save much time.

A final issue we needed to address was how the central management component would start the deployment of the remote components, which are described in Smartfrog description files, as we said. Clearly, this is a designing consideration and must be done at an early stage, because it affects the design of our system and the choice of the

communication mechanism we will describe next. Researching the Smartfrog API we found that Smartfrog requires our management console to be a Smartfrog component as well, if we want to start a Smartfrog description file from inside its class.

D. Communication mechanism

The communication mechanism between the management console and the remote components is a fundamental part of our system. As we have shown above, our design for the GUI requires the flow of messages to be one-way. Only the remote components are allowed to send messages to the GUI and not vice-versa.

The question we had to answer next was how we would build this mechanism. Obviously, we could choose some of the usual ways of communication between distributed components, such as RMI or plain sockets. The matter is that we use Smartfrog as our deployment framework. Hence we could exploit its capabilities. Smartfrog is built on RMI model, but goes well beyond that in many aspects. It provides a communication mechanism based on events. Currently, the events are allowed to be only strings and the mechanism is very simple and does not cover many needs. For more information about the 'workflow mechanism', as it is called, and the respective components, the reader may refer to the Smartfrog workflow document (Hewlett-Packard Development Company 7 Jul 2004, p 1-4).

Since the possible messages the remote components can send are a lot and must contain some common details, we wanted to create a standard format for them, such as if we were going to design our own protocol. The GUI could then understand the information each message carries more easily. The format we chose is the following:

0 or 1 (string) :: Name or IP of remote host
Message body

The initial 0 or 1 denotes whether the remote component checks the resources or is a deployment component. Then a string gives the name or the IP address of the remote host, so that the GUI knows where each message comes from. The message body starts in a new line.

8.2.2 Illustration – UML Diagrams

In the same way as in the elaboration phase, we will first present a component diagram that shows the high level view of our final system. We can see which high level components are deployed on the central host and on a random remote server and how they interact with each other.

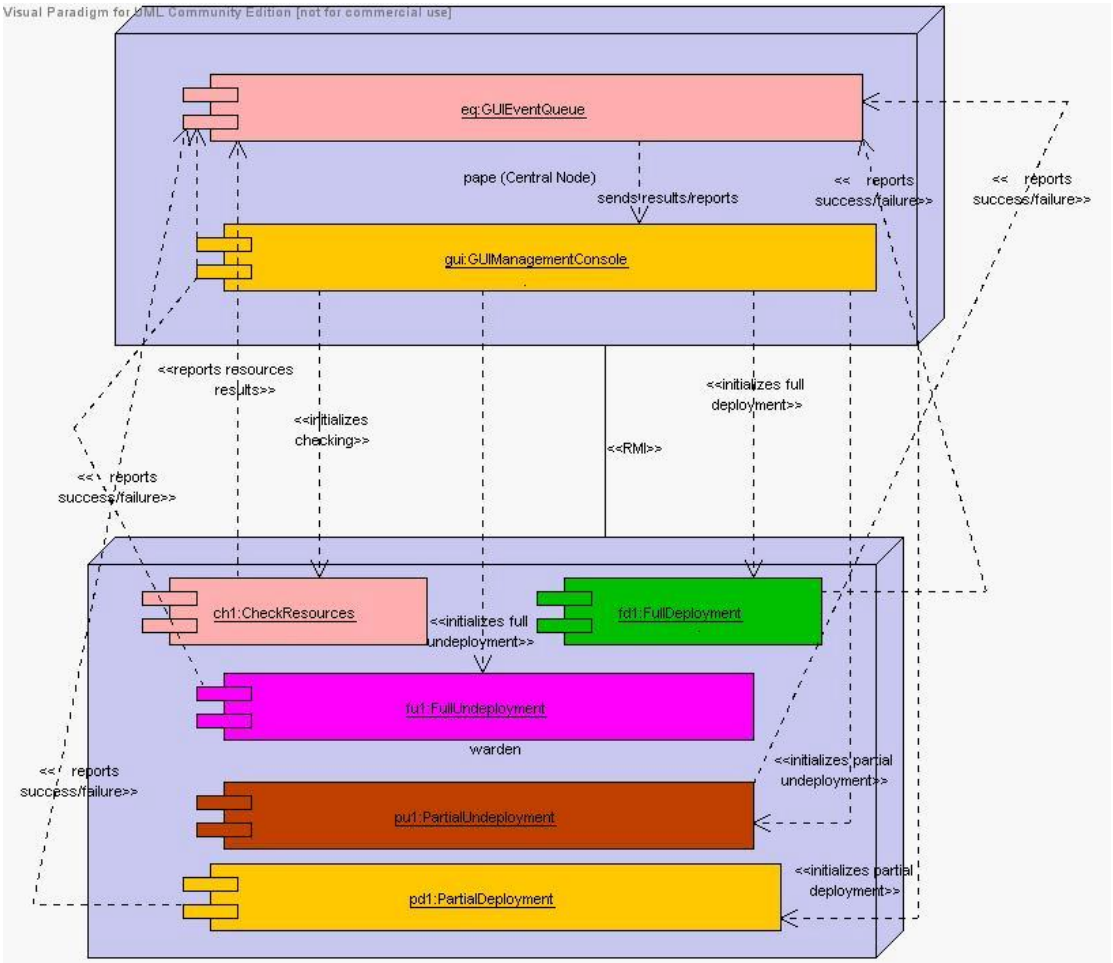

Figure 5

Next we will give the detailed diagrams for our system. The Smartfrog component descriptions will be treated as classes and will also be described with class diagrams in the developer's index. The different activities taking place during the deployment of the

Smartfrog components are described with activity diagrams. As the number of the classes and the components of our system are very large, we choose to split the class and the activity diagrams for the different parts of our system, so that we can show the functionality of each part in detail.

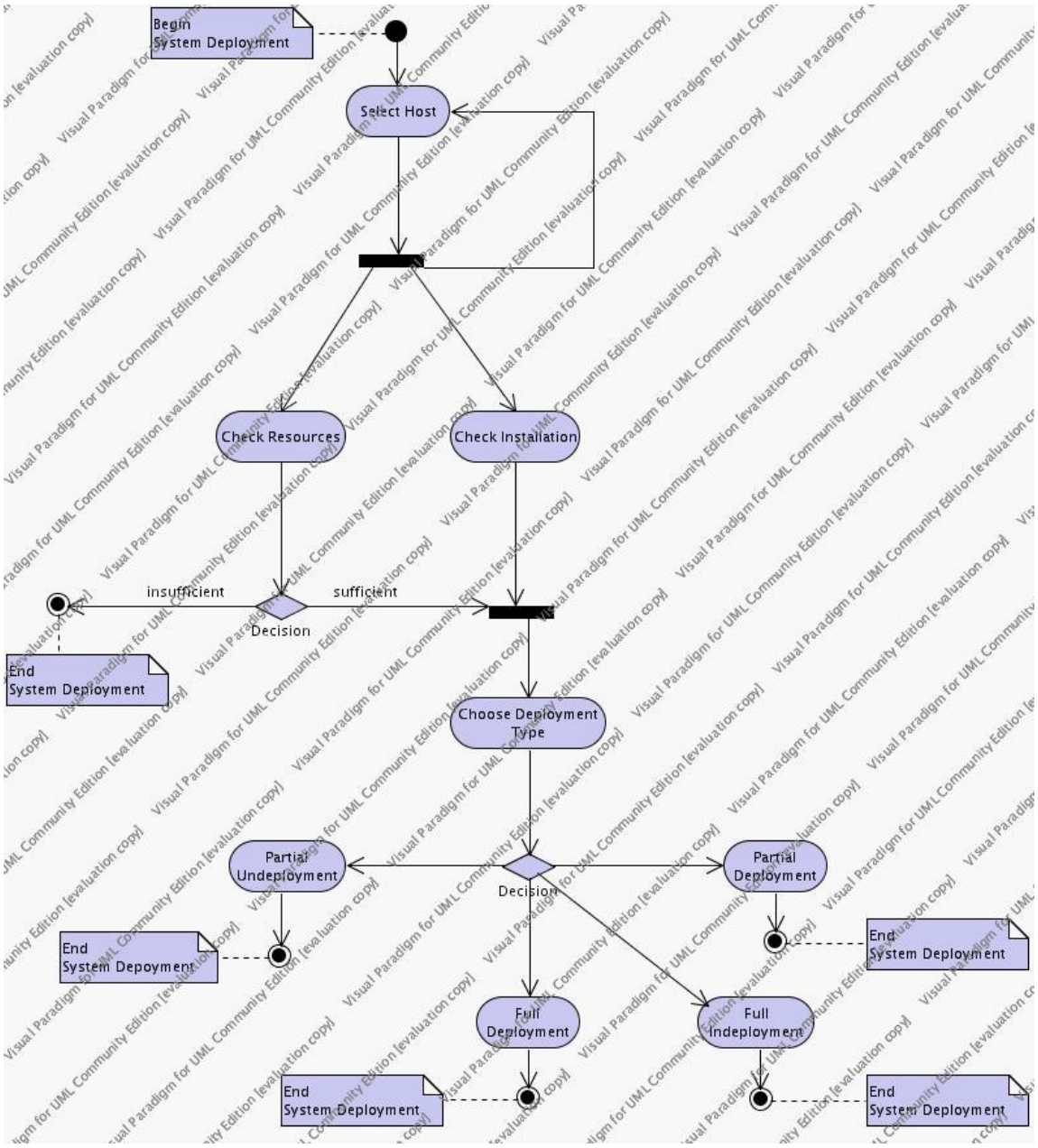

Figure 6: ActivityDiag-01

The *ActivityDiag-01* provides an overall view of our deployment system. It encompasses the major action components designed to provide the required basic functionality. The system starts by allowing the user to choose one or more hosts on which to deploy the

desired number of Grid Services. As the diagram indicates the deployment of the services is undertaken as an independent process for each chosen host. An autonomous component on each host checks the availability of the minimum necessary resources as well as the presence of pre-deployed infrastructure. The subsequent decision about the type of the deployment depends on the availability and presence of those pre-deployed resources. This design logic allows the user to interrupt the deployment process on a given host and proceed at a later time as well as it gives them the ability to structure the grid service deployment as an incremental process addressing a number of different scenarios.

The first of these scenarios occurs with 'clean' hosts – hosts on which there have not been deployed any of the necessary resources. The user is given the option to do a *Full Deployment*. The following diagram *ActivityDiag-02* illustrates a step down the hierarchy from the previous overview level. The process is logically divided in two sub-processes – *Deploy Base* and *Deploy Tier*. The analysis of the project requirements concerning the actual grid services installation reviled what specific files, applications and resources were necessary for their deployment and the steps preceding the deployment of the OGSA onto a Web Server (Apache Tomcat in our case) installation.

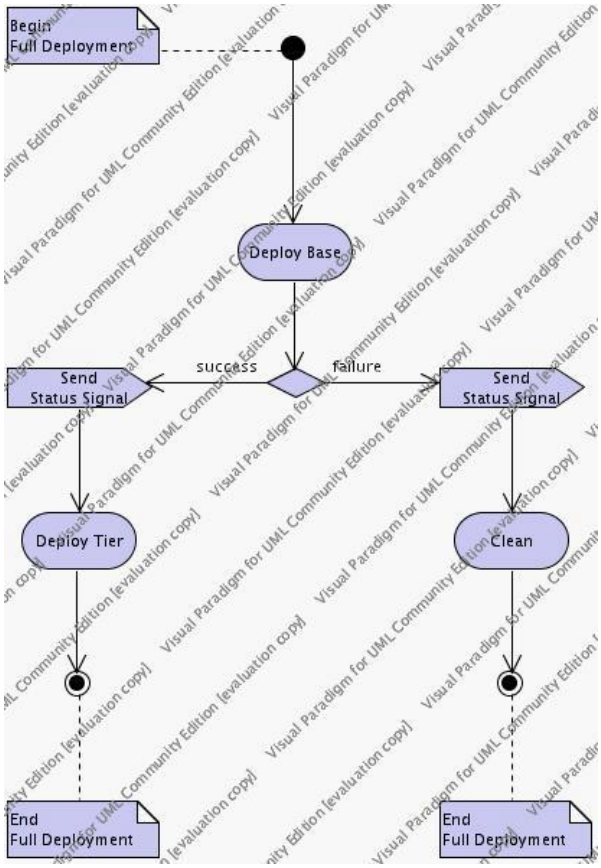

Figure 7: ActivityDiag-02

The download and installation of these basic resources is grouped in the *Deploy Base* action while the loading of the OGSA structure onto Tomcat's installation (as a Web application) and the activation of the Web Server is grouped in the *Deploy Tier* action component. In case of a failure after or during the *Deploy Base* action the application automatically starts *Clean* sub-process, which is designed to undo/remove *Deploy Base* actions. This 'roll-back' logic is incorporated in order to enforce all-or-nothing rule guaranteeing that no deployment is to proceed without the minimum of resources successfully installed on the host.

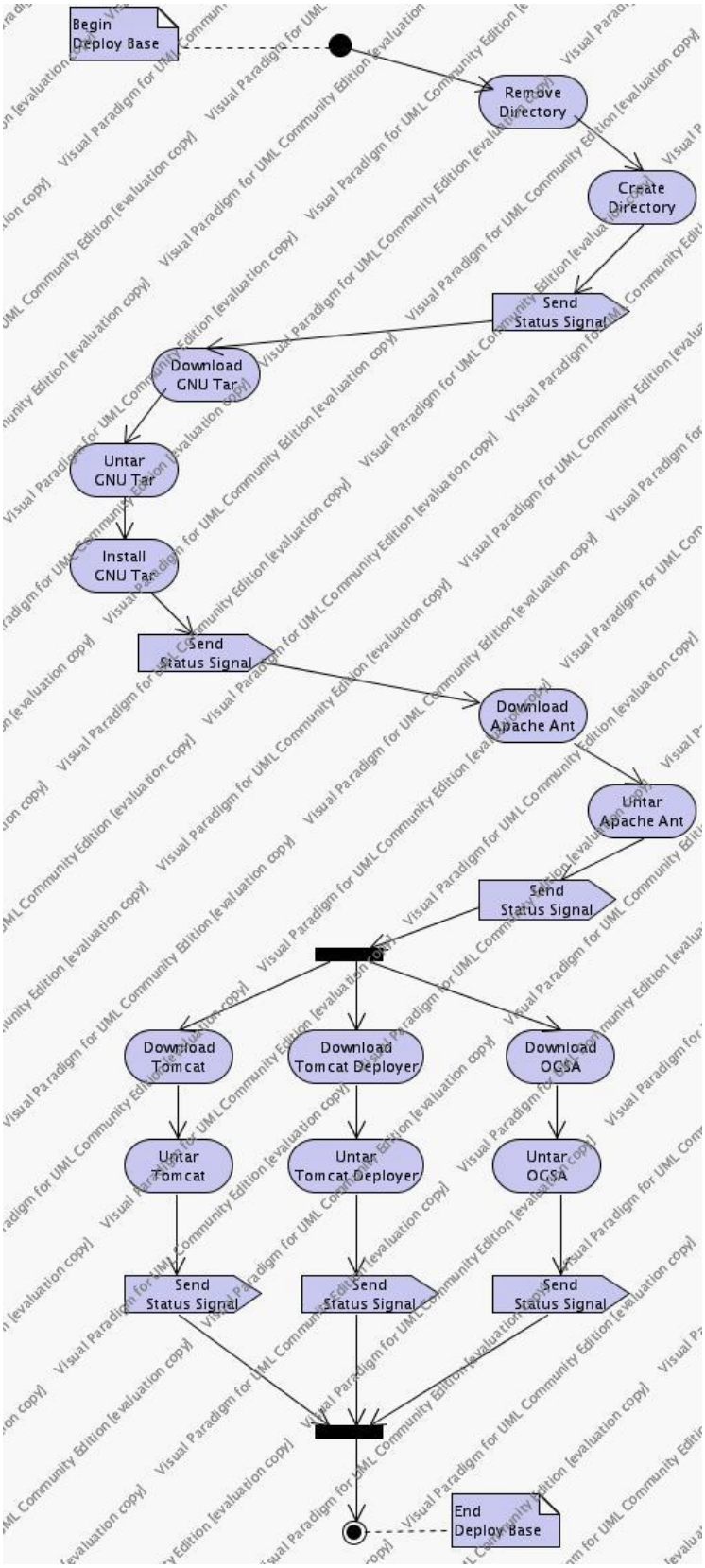

Figure 8: ActivityDiag-03

ActivityDiag-03 reveals further the details incorporated in the *Deploy Base* action group.
This is a sequence of action each resulting in a required resource being downloaded and installed. Following the steps on the diagram; the first step is removing and recreating of file directory structure where the actual installation of required deployment resources takes place. The status of this step as well as the statuses of the subsequent steps are sent to the User Interface for run-time monitoring of the progress. A failure in this or subsequent steps will trigger the automatic 'roll-back' action described in the previous paragraph. The next step is the download, decompression and installation of the GNU version of the Unix based Tar utility. This specific application and this version are required for the decompression of the subsequent resources that stipulate it in their release documentation. The next step in the sequence is the download and decompression of the Apache Ant utility. Ant is principle way of the deployment of Web applications and respectively Grid Services applications (which are base on Web Services infrastructure). The process makes extensive use of XML scripts provided with the installation packages and written under Ant specifications.
After the installation of Ant, the next three actions do not depend on the execution of each other, therefore, they are started in a parallel fashion. All three threads download and decompressed the installation packages of the Apache Tomcat, Tomcat Deployer and GlobusToolkit3.2. Just like the steps in the sequence before, the exit status of each thread is sent to the User Interface and a failure with each one causes the same 'roll-back' action described in the above paragraph. A successful exit status of the *Deploy Base* action as illustrated in *ActivityDiag-02* triggers the next action in the sequence at the upper hierarchical level – *Deploy Tier* in *ActivityDiag-04*.

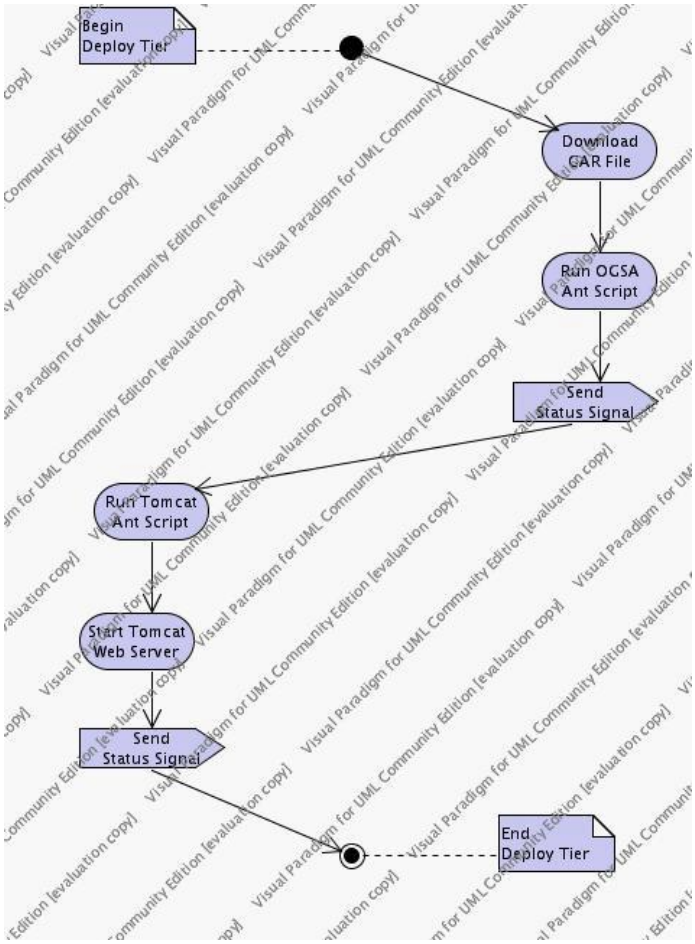

Figure 9: ActivityDiag-04

The *Deploy Tier* module is yet another sequence of actions that build on the resources deployed by *Deploy Base*. The first action of the sequence is the download of the Grid Service Archive (GAR) file and running the Ant scripts included in the GlobusToolkit3.2 installation for deploying Grid Services in the OGSA directory structure. The second action uses similar Ant scripts to deploy the OGSA directory structure as another Web application onto the Tomcat Web server and start up the server by running a provided for this purpose Unix shell script. The successful execution of these actions results in the Grid Service being effectively deployed onto a Web server and made accessible to client requests. The exit status of the sequence's step is again sent to User Interface for monitoring.

The activity diagrams so far illustrate the core functionality of the system – deploying grid services along with all necessary resources on a given host chosen by the user. The process is repeatable and incremental and it is independent of other processes running on other deployment hosts.

Based on the architectural design described in the preceding pages at the end of the phase, an executable architecture prototype was built in two or three iterations. While an evolutionary prototype of a production-quality component is always the goal for the Elaboration phase, our development process included one exploratory, throw-away prototype which was used for early demonstrations and helped us mitigate specific risks such as design/requirements trade-offs, component feasibility study etc. That first prototype reflected the initial project requirements that subsequently changed.

During the construction phase, all remaining components and application features were developed and integrated into the existing prototype, and all features were thoroughly tested. During the development process in this phase, more emphasis was placed on managing resources and controlling operations to optimize feature details, schedules, and quality. The way we structured our project allowed us to spawn parallel construction increments. These parallel activities significantly accelerated the availability of deployable releases. During the construction phase, an expanded version of the previous phase prototype was iteratively and incrementally developed and made ready for a final demonstration. This included finishing the implementation and testing of the software as a complete system from a user prospective. The additions made to the earlier prototype include, in terms of functionality, the incorporation of signal passing scheme for inter-component communication and tying the deployment process flow to the type of those signals passed as well as the capability of the system to handle 'scenarios' where only partial deployment (full deployment is covered in the preceding section) and un-deployment or full or partial de-installation was needed.

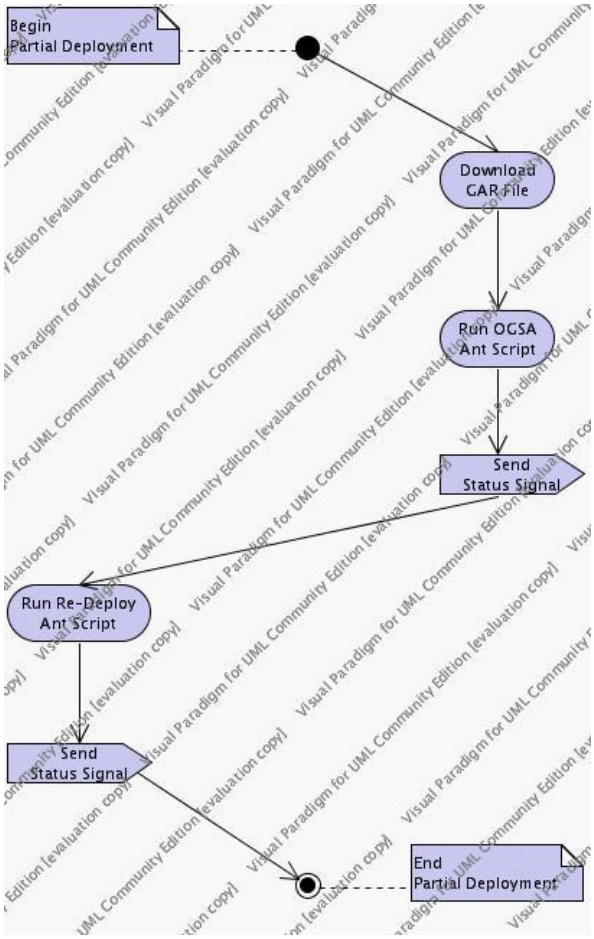

Figure 10: ActivityDiag-05

ActivityDiag-05 illustrates one such scenario. The diagram is an expanded view of the *Partial Deployment* action in *ActivityDiag-01*. The user is allowed to choose this option only if the previous actions *Check Resources* and *Check Installation* confirm the availability of sufficient computing resources and the presence of a pre-installed base of components (a result of successful *Full Deployment* execution, for example). In this case the user is given the ability to add another Grid Service to the group of services already provided by the OGSA installation on a given deployment host (see *ActivityDiag-04*). This process starts by downloading the GAR file from a location determined by the user to the project installation directory on the deployment host. At the next step an Ant script, already included with the OGSA installation package, is executed to deploy the new grid service onto the OGSA's infrastructure. A successful exit status triggers the third step that in turn executes an Ant script, included with Tomcat installation package, 'hot' re-loading OGSA into Tomcat. The use of this 'hot' re-loading feature eliminates the need of shutting down and re-starting the web server. Such start-stop downtime would cause disruptions of the other already running services and would result in failures of those services' client

requests. The two *Send Status Signal* actions are part of the signal passing scheme mention in the previous paragraph providing run-time status information to the User Interface. The successful completion of the process covered in this diagram results in the grid service chosen by the user being made available for client requests.

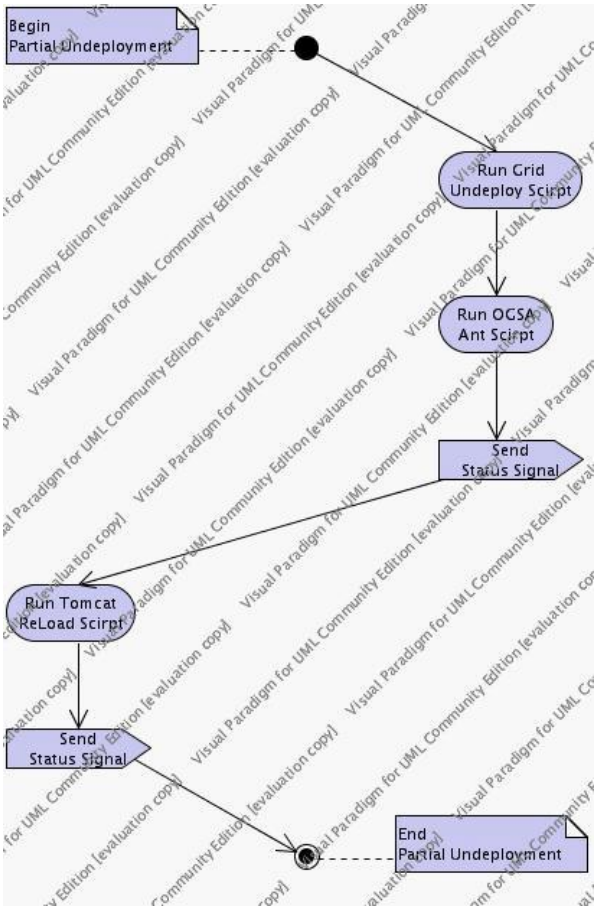

Figure 11: ActivityDiag-06

ActivityDiag-06 illustrates another scenario the Elaboration phase prototype was expanded to handle. The diagram is an expanded view of the *Partial Undeployment* action in *ActivityDiag-01*. The user is allowed to choose this option only if the previous actions *Check Resources* and *Check Installation* confirm the availability of sufficient computing resources and the presence of a pre-installed base of components (a result of successful *Full Deployment* execution, for example). In this case the user is given the ability to remove a Grid Service from the group of services provided by the OGSA installation on a given deployment host (see *ActivityDiag-04*). The process is a reverse of *Partial Deployment* described in the previous paragraph. It starts by executing an Ant script, already included with the OGSA installation package, that un-deploys the grid service from the OGSA's

infrastructure. A successful exit status triggers the second step that in turn executes an Ant script, included with Tomcat installation package, 'hot' re-loading OGSA into Tomcat. The use of this 'hot' re-loading feature eliminates the need of shutting down and re-starting the Web server. Such start-stop downtime would cause disruptions of the other already running services and result in failures of those services' client requests. The two *Send Status Signal* actions are part of the signal passing scheme providing run-time status information to the User Interface. The successful completion of the process covered in this diagram results in the grid service chosen by the user being made no longer available for client requests.

* For the purpose of keeping this report concise all UML Class diagrams illustrating the components providing the actual functionality of the Activity diagrams on the preceding pages have been moved to the Developer's Section document.

8.3 Implementation

Once again, we will discuss the implementation issues and our decisions for each one of the four building blocks of our system that we introduced in the design section.

8.3.1 Deployment of the infrastructure and the grid service

For this part we had to create the following components:

- deployment of the infrastructure
- deployment of a grid service
- redeployment of a grid service
- undeploy a grid service
- undeployment of the infrastructure

Combination of one or two of these components corresponds to the functionality expected by the management console for the remote deployment components. For example, combination of the deployment of the infrastructure and the deployment of a grid service would result in full deployment, whereas partial deployment would be implemented using either the component for the deployment of a grid service or for the redeployment of a grid service, depending on whether OGSA is running or not.

Now let's see what is needed to implement each of the five components mentioned above. The most complex of all is the deployment of the infrastructure by far. The infrastructure consists of several parts, namely GNU Tar, Ant, Globus Toolkit and Tomcat. To deploy each of these parts there is a common sequence of actions that must take place; the respective tar.gz file must be downloaded from the Internet, then untared and unzipped, and finally a script which will complete the installation possibly needs to be executed. After the support software is installed, OGSA is deployed in Tomcat by calling a simple Ant script.

The deployment and the undeployment of a grid service are also done with a simple Ant script. For the undeployment we just need to delete the whole directory where the infrastructure is installed. Redeployment of a grid service happens when we deploy a grid service in OGSA and then redeploy OGSA in Tomcat while Tomcat is running. The key issue here that we don't want to shut down the web server and then restart it; instead, we want to force Tomcat to take the redeployment of OGSA (and, consequently, of the new grid service) into account, while it is running. After thorough research into Tomcat's capabilities and documentation, we found out there is a tool called Tomcat Manager, which allows for the management of the web applications running under Tomcat in real time. Tomcat Manager needs to be configured, so that the users that can make the changes and their permissions are set beforehand. Fortunately, the installation package of Tomcat comes with an Ant script, which allows for the easy configuration of Tomcat Manager.

The next thing we had to consider for the implementation of this part was how we could implement a sequence of actions in Smartfrog descriptions. Note that this is absolutely essential, because the deployment of each one of the five Smartfrog components mentioned above is broken down to a sequence of sub actions. Smartfrog provides that functionality through the use of the workflow component Sequence. Sequence defines a list of subcomponents to be executed in order of description. The second subcomponent starts only after the end of the lifecycle of the first component and so forth. An important characteristic of all the workflow components is that when one of the subcomponents fails and terminates abnormally, the whole workflow parent component terminates as well. For example, when one of the subcomponents taking part in the sequence of actions for the deployment of the infrastructure fails, the whole deployment terminates automatically, which is very helpful in our case. Furthermore, we may define a specific behavior when a Sequence component terminates abnormally (or normally), by wrapping the whole Sequence in a workflow Try component and by catching its exit status. This is

particularly useful when we have to think ways of sending messages with the result of the deployment back to the central management console.

Another implementation issue is related to the fact that a lot of our components share some common functionality or are dependent on each other. As we said, the deployment of the various parts of the infrastructure is broken down to the same sequence of actions. The best way to take advantage of this feature is to create a basic component (called 'installer' or 'deployer') which incorporates this functionality and then make all the other components extend this component by overriding some of the attributes or adding new attributes and subcomponents where required.

In general, it is a very good programming technique in Smartfrog to create some initial base components and make the subsequent components inherit from them using the parameterization mechanism of Smartfrog. We tried to apply this principle as broadly as possible. On top of the hierarchy we created files with global configuration, which affect the whole of our descriptions, such as names for the installation directories and the files to be created during the installation and so forth. In addition, we created a file containing all the base components needed for the installation. Finally, we described each one of the executable components mentioned at the beginning of the section in a separate file making sure that they extend and override the behavior of the base components as required to implement the desired functionality. The greatest advantage of this technique is that if we want to make a change into the behavior of one based component, we only need to make it in one place, but it affects all the executable components that depend on it automatically; just like the inheritance mechanism in OOD.

In the description of our implementation so far we have used a top-to-bottom way of explaining the hierarchy of our components. First we explained how the executable components are expressed as a sequence of actions and then how the subcomponents, which form that sequence, extend our base components, which is described in a different file. This method is continued until the lower levels of the hierarchy. The base components that lie on the bottom of the hierarchy are:

- downloader; used to download a file from a URL in the Internet
- untar; unzips and untars a file
- run a command; runs an operating system (usually shell) command. It is used in various cases, such as the creation of the installation directories and the execution of Ant or shell scripts.

Therefore, the final question we had to answer was how we were going to implement these components. Everything else could be described in Smartfrog description files using the inheritance mechanism we mentioned before. For these base components we had two alternatives. We either had to write our classes to implement their behavior or we could use Smartfrog components that achieve the same functionality. Fortunately, Smartfrog deployment framework includes such components, because they are very useful in several deployment occasions. Downloader, Untar and RunCommand components covered all of our needs.

8.3.2 Resources Check

The major challenge for this part of our system was how we going to measure/check the resources. In this case, Smartfrog does not include any components/classes offering this functionality, so we had to create our own special classes.

The available virtual memory of the Java Virtual Machine could easily be retrieved using a simple Java instruction. However, in order to measure the CPU load and the free hard disk space we had to use command line scripts. There were two issues we had to consider here:

- how we were going to execute commands from inside a Java class
- how we were going to deal with the inherent heterogeneity problem due to the dependence of the commands on the underlying operating system.

For the first issue we created our own utility class, which takes a command as an argument and executes it, while it also creates two threads, which print the output and error streams of the command. Thus we could see the exit value of the command, the output results and possible error messages. We chose to write our own customized class and not use the RunCommand component as before, because we want to add some more customized behavior in it, which had to do with the way it would interact with the main CheckResources class and how it would notify the latter class of the results.

As far as the second issue is concerned, we explained earlier that our main targets were UNIX-like systems. For that reason, we used UNIX commands and scripts. Beyond that, we included some lines of code in our class to check the type of the operating system, so that the addition of a Windows version of the commands and scripts could be done more easily. For the check of the CPU load we used the command ‘uptime’, while for the check of the free hard disk space we wrote our own script. The problem was that there is

no standard UNIX command that returns the load of the CPU as a % percentage. We regarded the command 'uptime' as the closest and the easiest way to measure the CPU load, although what it shows in reality is the average number of jobs in the CPU queue and not the average CPU load, as we would like. The script for measuring the free hard disk space, on the other hand, functioned very well and returned the exact arithmetic result we wanted.

Finally, part of the functionality of the CheckResources class was to check whether OGSA (and consequently Tomcat) is running. Reading Tomcat documentation we concluded that the most secure way to implement the check was to open the URL of OGSA with a browser and see if the servlets and the web applications are there. Obviously, in a java program we could not open a browser, but rather create a URL connection and check if the URL exists and contains what we expect.

8.3.3 Management Console

We believe there is no point in expanding on the implementation of the GUI. The reason for that is that the way we chose to present the buttons, the windows, etc is not that important for the functionality of our project. We described the capabilities of the GUI in the design section. We will only say that we used the Java Swing package. In general, GUI development does not have surprises. The tools you have are quite standard; you know what you want to create, so it's just a matter of putting the effort and the time to do it. However, GUI development is very time-consuming. We needed to write a lot of classes and devote a substantial amount of time to create a user-friendly GUI, which had the functionality we set in the design.

A fundamental part of the implementation of the management console was related to the underlying functionality that launched the remote components. Under real circumstances, the management console must be capable of managing the deployment on a large number of servers. Thus it is not reasonable as well as efficient to have the execution of the main program stopped each time a remote component has to be launched. For that reason, we decided to make our console spawn different threads that will be responsible for instructing Smartfrog to deploy the components described in the proper description file. In addition, we wanted to exploit the observation that the implementation of the deployment of the remote components from a java program is done in a common way for all the threads and for all the description files. Hence, we included this common behavior in a class called RemoteHostDeployer.

An important issue that concerned us when implementing that class was how we were going to make dynamic changes to Smartfrog component descriptions. The way Smartfrog deploys components described in description files is the following:

- the file with the component descriptions is parsed
- Smartfrog components (RMI objects) are created in memory
- they are deployed by the Smartfrog runtime environment based on the characteristics (values of attributes) of their descriptions.

An attribute of an already parsed Smartfrog component can change dynamically by using specific methods of the Smartfrog API.

The attributes we wanted to be defined dynamically by the management console and could not be hardcoded in the description files are the following:

- hostname: the name of the remote host the particular on which deployment will take place
- messages event queue: the name of the EventQueue component where the messages would be sent and stored before being consumed by the management console (see subsection D below)
- gar URL: the URL for downloading the gar file representing the grid service to be deployed
- gar ID: the name of the gar file; it is used during the undeployment of the grid service

The problem in our case is that many attributes in our static component descriptions depend on the above attributes. In the same way, many attributes depend on the attributes of the first level of the dependency tree and so forth. The dependencies reach very deeply in this tree. Obviously, we had to change the values of the attributes in this tree dynamically as well, because they had been statically defined before. The number of changes would be very large, which renders that solution impractical and inefficient. Although Smartfrog provides a mechanism that enables you to specify that the value of an attribute will be defined at run-time and after the parsing, the need for successive changes along the dependent attributes would still exist. After trying all the possible ways to deal with this issue, we ended up writing the values of those three attributes in a new description file just before parsing. It was the easiest, most effective and probably the only trick we could use. This way we were sure that the dependent attributes would have

the correct values even just after the parsing and the deployment could flow with no further problems.

A last issue that concerned us during the implementation of the management console was how we would store the server profiles in a file. We chose the serialization of objects in Java as the best to do it. At the start-up the GUI reads the file and deserializes the object to retrieve the server profile information. We only had to make that the class, which contains the server profile, could be serialized indeed, for not all Java classes are serializable.

8.3.4 Communication mechanism

As we said, we used the Smartfrog Event Mechanism to implement the communication between the remote components and the management console. To send a message a remote component uses an EventSend component, whose purpose is exactly that; to send a message. All Smartfrog workflow components that implement certain interfaces can send or receive messages, as long as they are registered to a component, which will send them the messages, or that component has a certain attribute that specifies the components it will send the messages to. In order to render our management console capable of receiving messages we just had to make it extend one of the Smartfrog workflow components (in our case EventCompound). Another component called EventQueue was needed to receive the messages from the remote components and store them, until the management console can consume them.

8.4 Testing

Just like it was done during the Elaboration phase the testing iterations during the Construction phase started as soon as the first module additions to the previous phase's prototype were developed. Unlike, however, those early tests that verified the basic functionality expected from the components, the new tests verified the synchronized work of the expanded system. The successful results of the tests verified the extra complexity incorporated guaranteed compliance with the project scope set at the beginning of the process and refined through the duration of it.

The testing of the software at this phase focused primarily on the complete system from a user prospective. The additions made to the earlier prototype include, in terms of

functionality, the incorporation of signal passing scheme for inter-component communication and tying the deployment process flow to the type of those signals passed as well as the capability of the system to handle 'scenarios' where only partial deployment un-deployment or full or partial de-installation was needed.

The following testing scenario and results are included here as an example of the testing techniques used during the Elaboration phase. To keep this report concise only selected samples of the testing sessions and truncated test result output are shown on the next pages.
All tests were performed on the following resources:

make: Dell Inspiron 5150
OS: RedHat Linux 9
CPU: 1.6GHz
RAM: 512MB
HDD: 10GB
Net: 10Mb LAN
IP: 128.16.80.24
SW: SmartFrog3.02.002.beta (updated with CVS developer version as of July 12 2004)
Sun's Java(TM) 2 Runtime Environment, Standard Edition (build 1.4.2_05-b04
** An instance of SmartFrog daemon was running during all tests.*

Make: SunBlade 100
OS: Solaris 8
CPU: 500MHz
RAM: 128MB
HDD: 35MB
Net: 10Mb LAN
IP: 128.16.9.178
SW: SmartFrog3.02.002.beta (updated with CVS developer version as of July 12 2004)
Java(TM) 2 Runtime Environment, Standard Edition (build 1.4.2_01-b06)
Java HotSpot(TM) Client VM (build 1.4.2_01-b06, mixed mode)
** An instance of SmartFrog daemon was running during all tests.*

All of the following tests were performed from an outside user's prospective. The first host *128.16.80.24 (vnovov-p.cs.ucl.ac.uk)* was used as a management console from which all deployment sessions were started, monitored and controlled remotely on the second host *128.16.9.178 (highpark.cs.ucl.ac.uk).* The test scenarios in this section centered on the framework structured in the design documentation. There are four primary modules corresponding to the four scenarios a user might encounter by running the system. Those are *Full Deployment*, *Partial Deployment*, *Partial Undeployment* and *Full Undeployment* (see Design section).

For the first scenario *Full Deployment*, the remote deployment host *128.16.9.178 (highpark.cs.ucl.ac.uk)* was checked for sufficient resources and the absence of any previous partial or full system deployments. The type of deployment is delivered by executing the modules in the *z15_4_SystemComponents.sf* on the management host. The test execution results in a full system deployment on the remote host i.e. All required files/application/utilities are downloaded, decompressed, installed and configured in the created for this purpose project directory */tmp/z15_4/* and respective sub-directories. The grid service specified in the *z15_4_GridArchiveFileNames.sf – gt3_asian_AsianSpreadOption* is deployed. Tomcat Web Server is started and facilitates access to that grid service. The screen shots below show the status of the remote system after this test was completed (the selected exerts from the log files have been truncated for clarity):

sfTrace: ROOT[3800]>HOST highpark.cs.ucl.ac.uk:rootProcess:full_deployment_test:deployBase:action:installationSequence1:deployProjectDir, DEPLOYING in rootProcess at highpark.cs.ucl.ac.uk/128.16.9.178:3800

. . .

sfTrace: ROOT[3800]>HOST highpark.cs.ucl.ac.uk:rootProcess:full_deployment_test:deployBase:HOST highpark.cs.ucl.ac.uk:rootProcess:full_deployment_test:deployBase_actionRunning sending 1::localhost systemBase-SUCCEEDED

. . .

sfTrace: ROOT[3800]>HOST highpark.cs.ucl.ac.uk:rootProcess:full_deployment_test:deployTier:action:deployGridSrvc, DEPLOYING in rootProcess at highpark.cs.ucl.ac.uk/128.16.9.178:3800

. . .

[z15_4BaseGarDeployer:runInstall(bash)] LOG > Executing: /tmp/z15_4.2004/apache-ant-1.6.1/bin/ant deploy -Dgar.name=/tmp/z15_4.2004/gar-files/gt3_asian_AsianSpreadOption.gar, SFRunShell, 2

. . .

sfTrace: ROOT[3800]>HOST highpark.cs.ucl.ac.uk:rootProcess:full_deployment_test:deployTier:action:deployWebSrvc:HOST highpark.cs.ucl.ac.uk:rootProcess:sending 1::localhost deployGridSrvc-SUCCEEDED

. . .

[z15_4WebServicesDeployer(bash)] OUT > deployWebapp:
[z15_4WebServicesDeployer(bash)] OUT > [copy] Copying 2 files to /tmp/z15_4.2004/jakarta-tomcat-5.0.25/webapps/ogsa

. . .

sfTrace: ROOT[3800]>HOST highpark.cs.ucl.ac.uk:rootProcess:full_deployment_test:deployTier:HOST highpark.cs.ucl.ac.uk:rootProcess:full_deployment_test:deployTier_actionRunning, sending 1::localhost systemTier-SUCCEEDED

A scan of the project directory */tmp/z15_4/* contents shows all necessary files and directories were present as well as the deployment of OGSA onto Tomcat directories along with the specified grid service *gt3_asian_AsianSpreadOption*:

```
/tmp/z15_4.2004/:
total 96
drwx------   6 vnovov   msc         950 Feb 12  2004 apache-ant-1.6.1
drwx------   2 vnovov   msc         205 Aug 27 17:18 gar-files
drwx------  11 vnovov   msc         939 Aug 27 17:15 jakarta-tomcat-5.0.25
drwx------   5 vnovov   msc         564 Aug 27 17:16 jakarta-tomcat-5.0.25-deployer
drwx------  14 vnovov   msc        2140 Aug 27 17:18 ogsa-3.2
drwx------  13 vnovov   msc        2496 Aug 27 17:12 tar-1.13
. . .
/tmp/z15_4.2004/jakarta-tomcat-5.0.25/webapps/ogsa/schema/gt3.asian:
total 16
drwx------   2 vnovov   msc         203 Aug 27 18:15 AsianSpreadOption

/tmp/z15_4.2004/jakarta-tomcat-5.0.25/webapps/ogsa/schema/gt3.asian/AsianSpreadOption:
total 16
-rw-------   1 vnovov   msc        3508 Aug 27 18:15 AsianSpreadOptionService.wsdl
. . .
/tmp/z15_4.2004/jakarta-tomcat-5.0.25/webapps/ogsa/WEB-INF/lib:
total 9792
-rw-------   1 vnovov   msc     1174629 Aug 27 17:19 axis.jar
-rw-------   1 vnovov   msc       16217 Aug 27 17:19 cog-axis.jar
-rw-------   1 vnovov   msc       13018 Aug 27 17:19 gt3.asian.AsianSpreadOption-stub.jar
-rw-------   1 vnovov   msc        3776 Aug 27 17:19 gt3.asian.AsianSpreadOption.jar
```

Using a Web Browser and the OGSA Service Browser utility it was verified that the grid service *gt3_asian_AsianSpreadOption* was successfully deployed and it was accessible for client requests on *128.16.9.178 (highpark.cs.ucl.ac.uk)*:

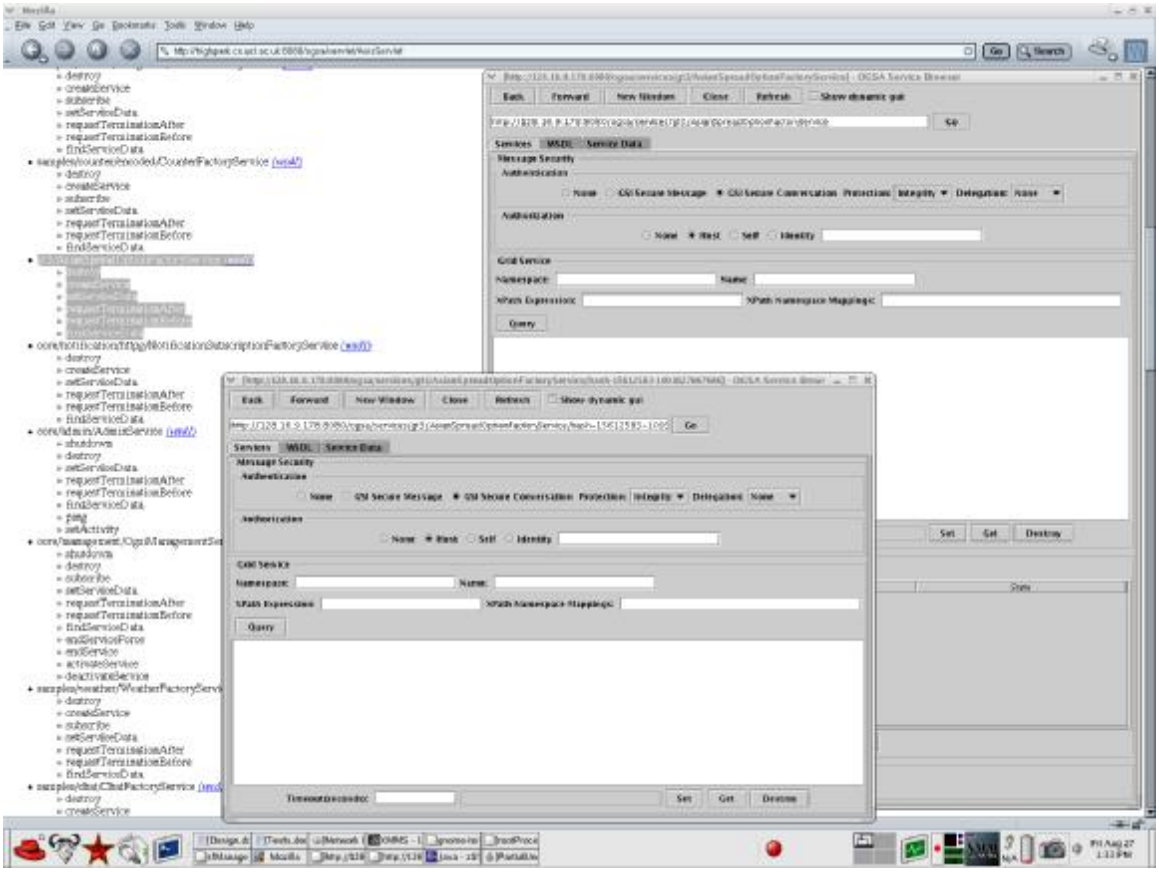

Figure 12

The second scenario tested was *Partial Deployment*. This type of deployment is delivered by executing the modules in the *z15_4_NodeSystemRedeployTier.sf* on the management host. The test execution results in a new grid service being deployed onto the OGSA's directories, OGSA being re-deployed as Web application directories inside the Tomcat Web Server installation (both deployed by the preceding tests) and Tomcat being 'hot' re-loaded at the end. The screen shots below show the status of the remote system after this test was completed (the selected exerts from the log files have been truncated for clarity):

sfTrace: ROOT[3800]>HOST highpark.cs.ucl.ac.uk:rootProcess:partial_deployment_test:deployGridSrvc, DEPLOYING in rootProcess at highpark.cs.ucl.ac.uk/128.16.9.178:3800

. . .

[z15_4BaseGarDeployer:runInstall(bash)] LOG > Executing: /tmp/z15_4.2004/apache-ant-1.6.1/bin/ant deploy -Dgar.name=/tmp/z15_4.2004/gar-files/gt3_european_StandardEuropeanOption.gar, SFRunShell,

. . .

[z15_4WebServicesReDeployer(bash)] LOG > Executing: /tmp/z15_4.2004/apache-ant-1.6.1/bin/ant redeployTomcat -Dtomcat.dir=/tmp/z15_4.2004/jakarta-tomcat-5.0.25/, SFRunShell, 2

. . .

[z15_4WebServicesReLoader(bash)] LOG > Executing: /tmp/z15_4.2004/apache-ant-1.6.1/bin/ant reload -Dpath=/ogsa, SFRunShell, 2

A scan of the project directory */tmp/z15_4.2004/jakarta-tomcat-5.0.25/* contents shows all necessary files and directories were present as well as the deployment of OGSA onto Tomcat directories along with the specified service *gt3_european_StandardEuropeanOption*:

/tmp/z15_4.2004/jakarta-tomcat-5.0.25/webapps/ogsa/schema/gt3.european:
total 16
drwx------ 2 vnovov msc 208 Aug 27 17:46 StandardEuropeanOption

/tmp/z15_4.2004/jakarta-tomcat-5.0.25/webapps/ogsa/schema/gt3.european/StandardEuropeanOption:
total 16
-rw------- 1 vnovov msc 3456 Aug 27 17:46 StandardEuropeanOptionService.wsdl

/tmp/z15_4.2004/jakarta-tomcat-5.0.25/webapps/ogsa/WEB-INF/lib:
total 9840
-rw------- 1 vnovov msc 1174629 Aug 27 17:46 axis.jar
-rw------- 1 vnovov msc 16217 Aug 27 17:46 cog-axis.jar
-rw------- 1 vnovov msc 13018 Aug 27 17:46 gt3.asian.AsianSpreadOption-stub.jar
-rw------- 1 vnovov msc 3776 Aug 27 17:46 gt3.asian.AsianSpreadOption.jar
-rw------- 1 vnovov msc 12917 Aug 27 17:46 gt3.european.StandardEuropeanOption-stub.jar
-rw------- 1 vnovov msc 3555 Aug 27 17:46 gt3.european.StandardEuropeanOption.jar

Using a Web Browser and the OGSA Service Browser utility it was verified that the grid service *gt3_european_StandardEuropeanOption* along with *gt3_asian_AsianSpreadOption*, was successfully deployed and it was accessible for client requests:

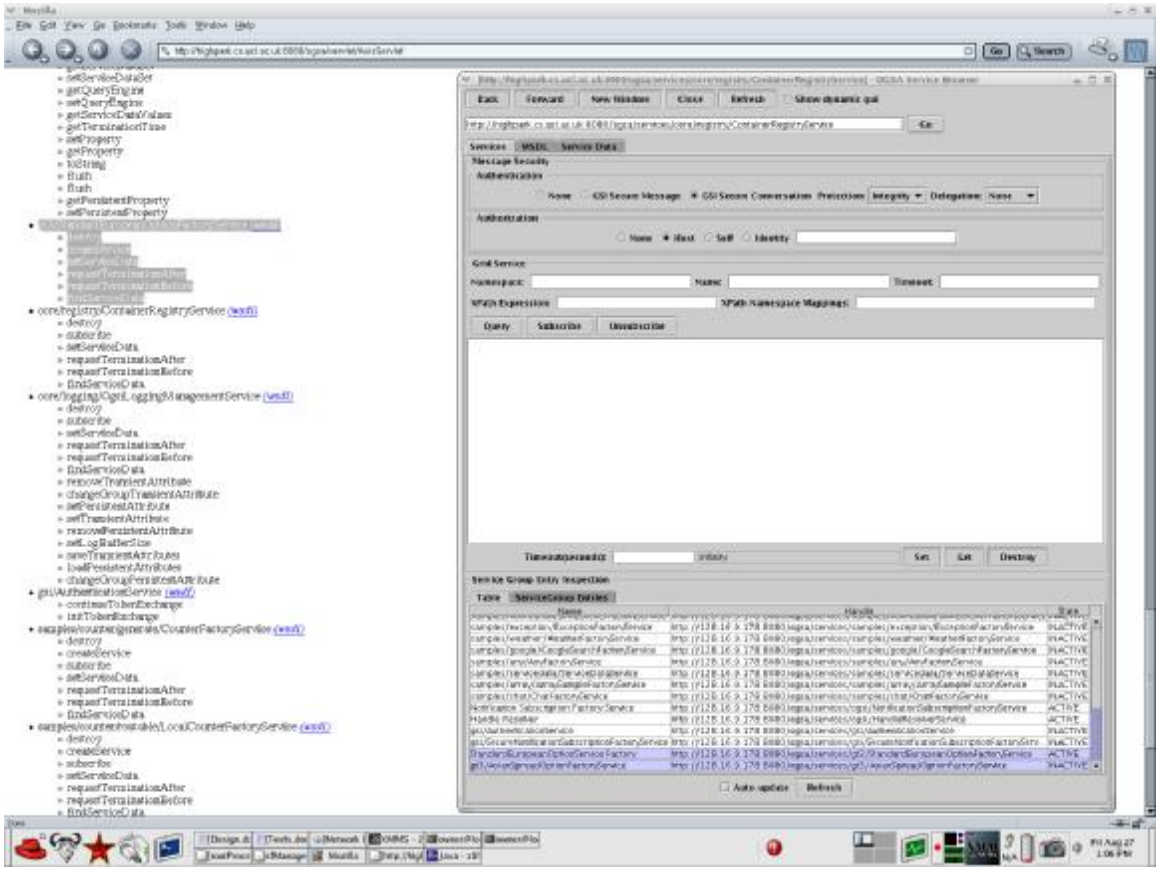

Figure 13

The third scenario tested was *Partial UnDeployment.* This type of deployment is delivered by executing the modules in the *z15_4_PartialGridServiceUnDeployer.sf* on the management host. The test execution results in a grid service being un-deployed from the OGSA's directories , OGSA being re-deployed as Web application directories inside the Tomcat Web Server installation (both deployed by the preceding tests) and Tomcat being 'hot' re-loaded at the end. That has an exactly reverse to the previous scenario effect. The screen shots below show the status of the remote system after this test was completed (the selected exerts from the log files have been truncated for clarity):

sfTrace: ROOT[3800]>HOST highpark.cs.ucl.ac.uk:rootProcess:partial_UNdeployment_test:unDeployGar:action, DEPLOYING in rootProcess at highpark.cs.ucl.ac.uk/128.16.9.178:3800

. . .

[z15_4BaseGarUnDeployer(bash)] LOG > Executing: /tmp/z15_4.2004/apache-ant-1.6.1/bin/ant undeploy -Dgar.id=gt3_european_StandardEuropeanOption, SFRunShell, 2

. . .
[z15_4BaseGarUnDeployer(bash)] LOG > Executing: /tmp/z15_4.2004/apache-ant-1.6.1/bin/ant redeployTomcat -Dtomcat.dir=/tmp/z15_4.2004/jakarta-tomcat-5.0.25/, SFRunShell, 2
. . .
[z15_4WebServicesReLoader(bash)] LOG > Executing: /tmp/z15_4.2004/apache-ant-1.6.1/bin/ant reload -Dpath=/ogsa, SFRunShell, 2

A scan of the project directory */tmp/z15_4.2004/jakarta-tomcat-5.0.25/* contents shows all necessary files and directories were removed as well as the deployment of OGSA onto Tomcat directories without the specified service *gt3_european_StandardEuropeanOption*:

/tmp/z15_4.2004/jakarta-tomcat-5.0.25/webapps/ogsa/WEB-INF/lib:
total 9840
-rw------- 1 vnovov msc 1174629 Aug 27 17:46 axis.jar
-rw------- 1 vnovov msc 16217 Aug 27 17:46 cog-axis.jar
-rw------- 1 vnovov msc 13018 Aug 27 17:46 gt3.asian.AsianSpreadOption-stub.jar
-rw------- 1 vnovov msc 3776 Aug 27 17:46 gt3.asian.AsianSpreadOption.jar

Using a Web Browser and the OGSA Service Browser utility it was verified that the grid service *gt3_european_StandardEuropeanOption* was successfully un-deployed and it was not accessible any longer for client requests on *128.16.9.178 (highpark.cs.ucl.ac.uk)*:

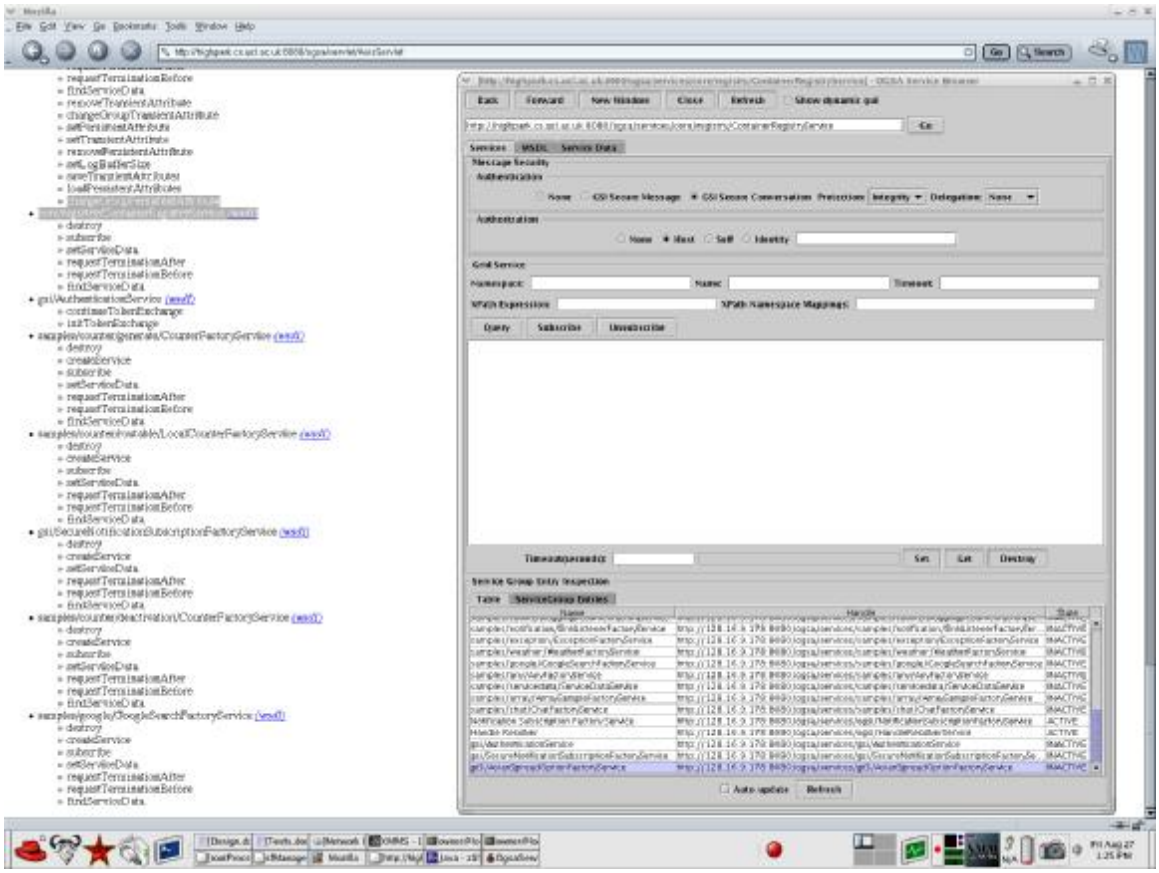

Figure 14

The fourth scenario tested was *Full UnDeployment*. This type of deployment is delivered by executing the modules in the *z15_4_FullInstallationUnDeployer.sf* on the management host. The test execution results in the whole installation in the remote deployment host being removed completely. The module execution shutdowns Tomcat Web Server and it then deletes the project directory */tmp/z15_4/* along with all its contents. The screen shots below show the status of the remote system after this test was completed (the selected exerts from the log files have been truncated for clarity):

sfTrace: ROOT[3800]>HOST highpark.cs.ucl.ac.uk:rootProcess:full_UNdeployment_test, DEPLOYING in rootProcess at highpark.cs.ucl.ac.uk/128.16.9.178:3800

. . .

[z15_4FullInstallationUnDeployer(bash)] LOG > Executing: /tmp/z15_4.2004/jakarta-tomcat-5.0.25/bin/shutdown.sh, SFRunShell, 2

. . .

sfTrace: ROOT[3800]>HOST highpark.cs.ucl.ac.uk:rootProcess:full_UNdeployment_test:action:removeInstallationDir, DEPLOYING in rootProcess at highpark.cs.ucl.ac.uk/128.16.9.178:3800

. . .

[z15_4BaseInstallationUnDeployer:removeInstallationDir(bash)] LOG > Executing: rm -Rf z15_4.2004, SFRunShell, 2

A scan of the directory */tmp/* where the project directory *z15_4.2004/* was created shows all previous files and directories were removed:

2 vnovov@highpark% ls -lR /tmp/z15_4.2003/
/tmp/z15_4.2004/: No such file or directory
3 vnovov@highpark%

Using a Web Browser it was verified that Tomcat Web Server was not available on the remote deployment host *128.16.9.178 (highpark.cs.ucl.ac.uk)*:

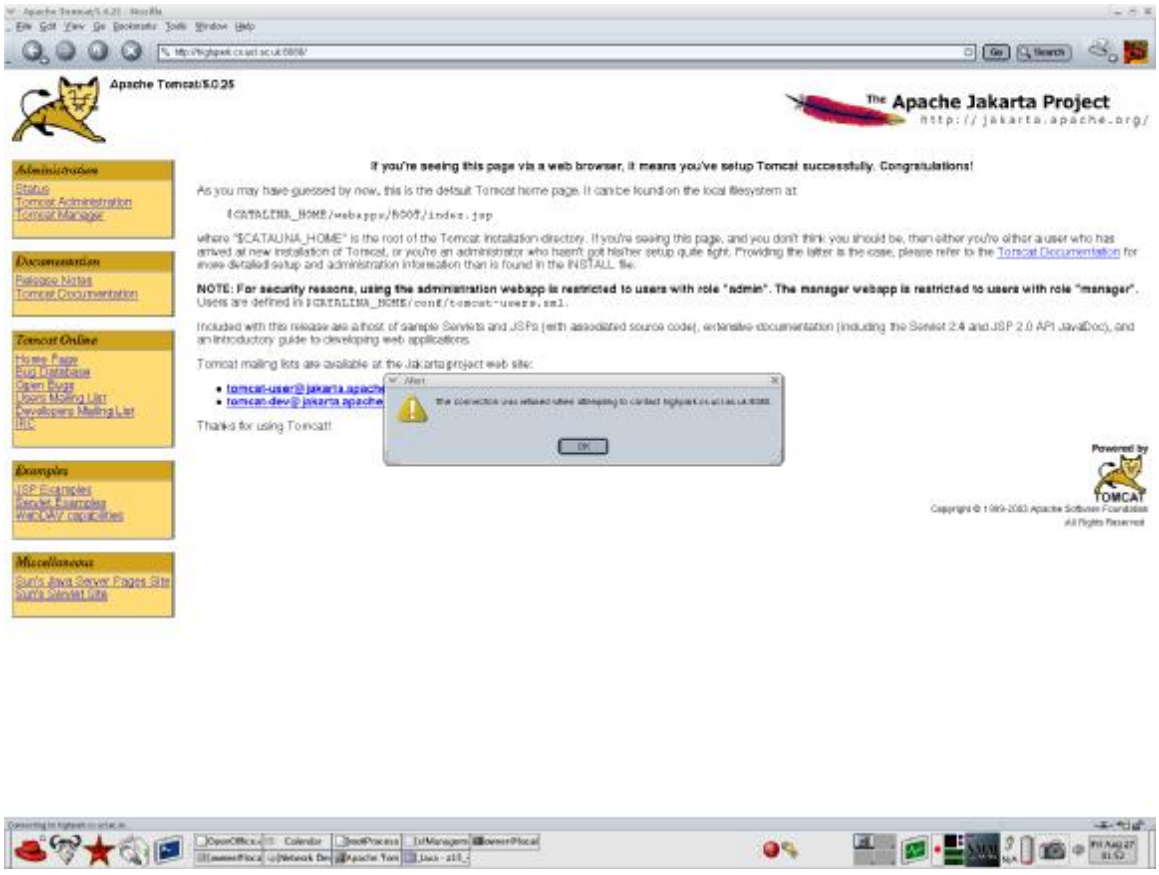

Figure 15

9 MEASUREMENTS

During the project many measurements have been taken and in this chapter they will be used in order to criticize the use of SmartFrog and its efficiency. The financial application that was created by Charaka Goonatilake has been used for all the measurements.

9.1 Measurements and conclusions about deployment

This part deals with the measurements that have been taken during the deployment of the grid services and their infrastructure with and without SmartFrog. It will include measurements of the traditional manual process, of the first incarnation of our system created during the elaboration phase and of the final incarnation of the construction phase. These measurements will allow us to figure out whether the use of SmartFrog speeds up the deployment process. Specific details will show what the difference is of using and not using SmartFrog and also whether there is any difference between the two incarnations of our system during the elaboration and the construction phase.

The platform that has been used for these measurements is the same as the one where the testing took place. The specs of the computers are the following:

Sunblade 100
SunOS 5.8
Solaris 8
500 Mhz
128 MB

It is important to take into consideration a couple of important factors:

- All the measurements have been taken on the same computers. Certain computers perform in a different way than others but that will not make any difference since the same computers have been used for all the different sets of measurements.
- The computers are all located in a lab. This means that it is not possible for us to calculate a very important overhead of the traditional deployment process. In another scenario the computers that will be used may actually be located far apart. In different buildings, different cities, even different countries. Therefore the administrator would

either have to visit these computers in other buildings are arrange for a cooperation with administrators in other cities and countries to do part of the deployment. In any case there is a great overhead that we will not take into account. It should be noted though that this overhead would certainly increase the value of SmartFrog as a deployment tool.

9.1.1 Traditional approach

It is quite certain that the traditional approach is lengthier than the system created with SmartFrog. However, in order to establish the exact benefit that SmartFrog provides it is important to measure the time that it takes to deploy grid services and their infrastructure in a manual way.

Note: The deployment of infrastructure and of the grid service is quite complicated. Before taking the actual measurements, the deployment process was practiced so that they would not include any time lost due to unfamiliarity.

Deployment of infrastructure: Tomcat and GT3 were downloaded from the internet and installed on ten machines. The simplest most basic configuration options were used. The process was done sequentially on each of the ten machines

Time: 12 minutes for each computer

Deployment of grid service: The grid service file (GAR) was transferred to a specific directory on each machine, it was deployed with Ant and then Tomcat was started. The process was done sequentially for each of the ten machines.

Time: 5 minutes for each computer.

Comment: It should be noted that during the 12 minutes, which are, needed for each computer the administrator can not take on other tasks. His full and continuous attention is needed in order to deploy the software.

9.1.2 Measurements of Elaboration

The SmartFrog components that have been designed and implemented take care of the entire deployment process. The only manual labor that the administrator must undertake is to start the SmartFrog daemon on each computer. The time that is needed to do that has not been included because it's possible to start the daemon with the operating system's start-up script.

Deployment of infrastructure and grid service: In order to begin the deployment a single line is used as a command, which runs the SmartFrog component. The infrastructure is deployed along with a grid service.

Time: 7 minutes for each computer

Comment: It is quite clear that SmartFrog allows for a much faster deployment. The gain in time is even bigger if one considers that more than one deployments can occur almost simultaneously. That will become clearer later on.

9.1.3 Measurements of Construction

Once again the SmartFrog daemon must be running on each computer that will be used. This time the administrator can use the GUI and control deployment on all machines from that.

Deployment of Infrastructure and grid service: The administrator simply has to start the GUI and use it to perform deployment and undeployment of infrastructure.

Time: 7 minutes for each computer

Deployment of grid service: This time there an extra option to deploy an extra grid service.

Time: 1 minute for each computer

Comment: It is important to mention here that the administrator does not actually need knowledge of SmartFrog's inner workings in order to perform the deployment. The messages that the GUI receives provide adequate information. Also, the measurements of construction are the same as the ones of elaboration. The reason is that in elaboration we tried to better manage and control the process rather than automate. Finally, during the deployment process the administrator does not need to pay any attention to the computers. The process is automated and the administrator will be alerted when something is wrong. So during the time of deployment he may deal with other tasks.

9.2 Measurements and conclusions about the use of SmartFrog

It became obvious, from the above measurements, that SmartFrog does indeed greatly reduce the time that is needed for deployment. However, the most important gain is the fact that the deployment process is now automated and can occur remotely which means that it can occur simultaneously. The following graph depicts the time that is needed to deploy the infrastructure up to 20 computers with and without SmartFrog.

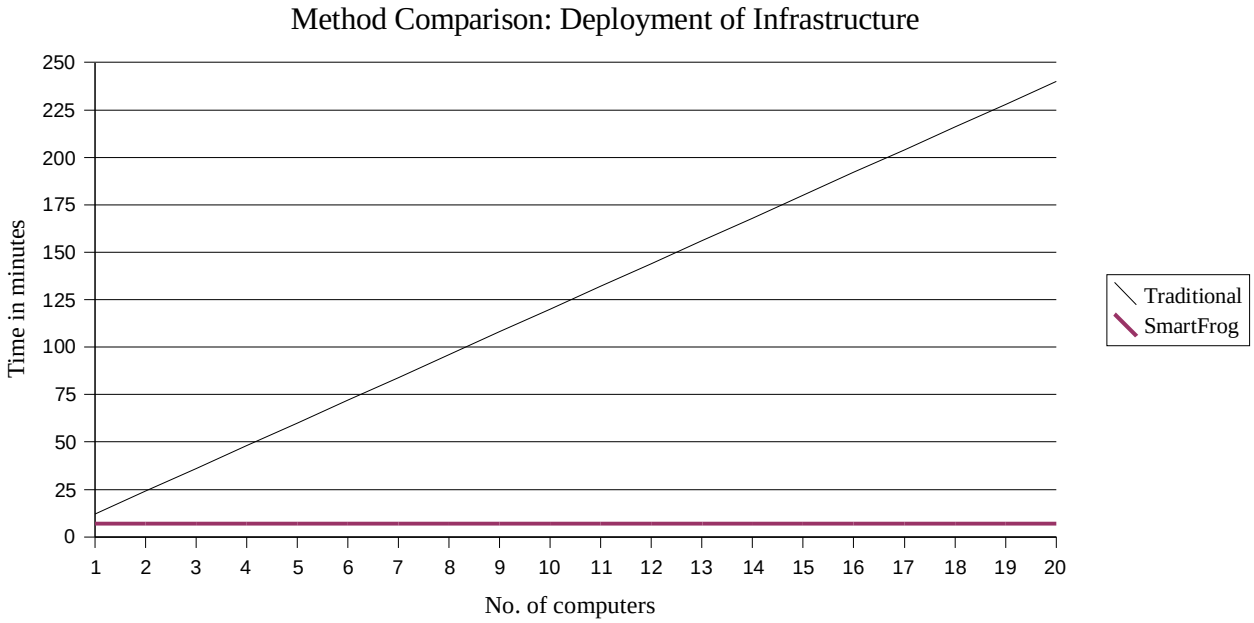

Figure 16

The slim line is the traditional method. It increases immensely because the deployments are done sequentially by one administrator. Similar is the chart for the deployment of a grid service.

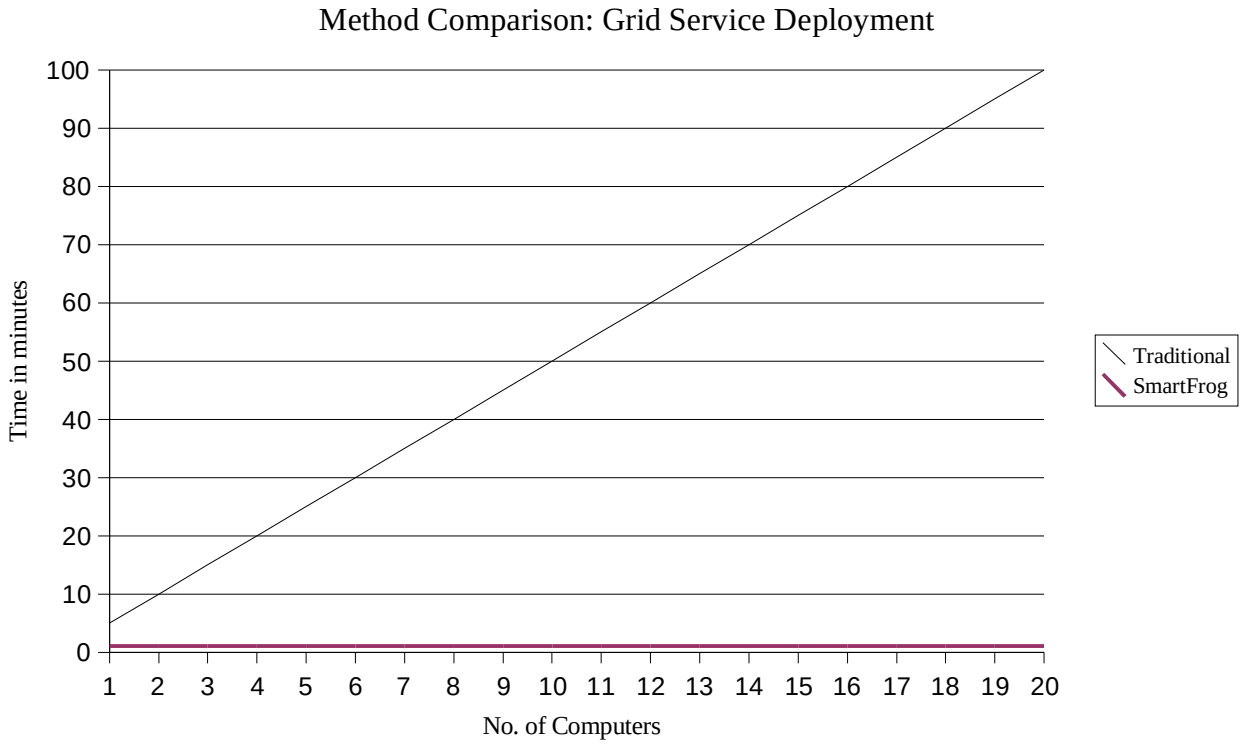

Figure 17

From the two graphs, figures 16 and 17, it is obvious that SmartFrog is a great asset, which can greatly reduce the time that is needed for deployment. The gain in time is related to the number of computers that will be used. So, for example, 100 computers must be used, SmartFrog would reduce the time from 1200 minutes to just 7, saving almost 20 man-hours. If one considers the possible updates and new grid services that may need to be deployed, it becomes obvious that SmartFrog can help.

We have now established that using SmartFrog as a tool to deploy grid services provides great advantages. However, we need to think about the effort that was needed in order to learn how to use it, and the effort that is needed in order to write the components to deploy a grid service. In order to do that we will use the notes that we kept throughout the project.

Important: The following calculations are arbitrary and are based on our own work. They are meant to give an idea about SmartFrog's ability and just that. We need to find out whether learning SmartFrog is worth the effort. In order to do that we will use our measurements. Those measurements depend fully upon our capabilities as individuals, we

quickly we managed to learn SmartFrog and how quickly we managed to design and implement the components.

We needed 160 man-hours to learn SmartFrog. That is, 160 man-hours per person. That is something that depends a lot on the individual's experience so it should be assumed that it would be very different for a senior developer than it is for a student. Also, SmartFrog is still in beta testing. It does not have adequate documentation and there are not many sources of information. In the future that will probably change. Moreover, after 160 man-hours our knowledge of SmartFrog was not complete. It had simply reached a point where we were able to start designing and implementing the components. Many gaps had to be filled along the way.

We needed 224 man-hours to design, implement and test the SmartFrog components during the elaboration phase. That is not an accurate and fair calculation because during these hours we were still learning about SmartFrog. During the construction phase the time that was needed was greatly reduced, 128 man-hours, which means that as our experience with SmartFrog increased, the time that is needed to write the components is reduced.

In order to make a meaningful comparison we need to have a scenario.

Important: Once again, many assumptions will made. We need a scenario in order to contradict what we did it (using SmartFrog) with what someone else would do (manual approach) in order to be able and calculate gains. The manual approach requires absolutely no work (learning, designing, implementing) beforehand so it's clear that it will have zero costs in the beginning. But, it will have high costs as far as the actual deployment is concerned.

One could say that the following formula gives the total time that will be spent when dealing with the deployment of grid services:

Total time spent = Time spent learning and implementing the deployment system + Time spent deploying the grid services over a length of time

We need to find out what the total time spent is for both approaches, SmartFrog and traditional.

So, a scenario is needed. This scenario needs to be close to the realistic needs of an administrator that is using grid services. We assume the existence of an administrator

who has a hundred computers running grid services. There are ten grid services running on each computer. The administrator must deploy the infrastructure and the grid services on each computer. He must also update the infrastructure four times every year because of updates of the programs. He must also update the grid services themselves five times per year.

In Figure 18 we have mapped the cost in man-minutes that is needed to deploy grid services against the length of time for both approaches.

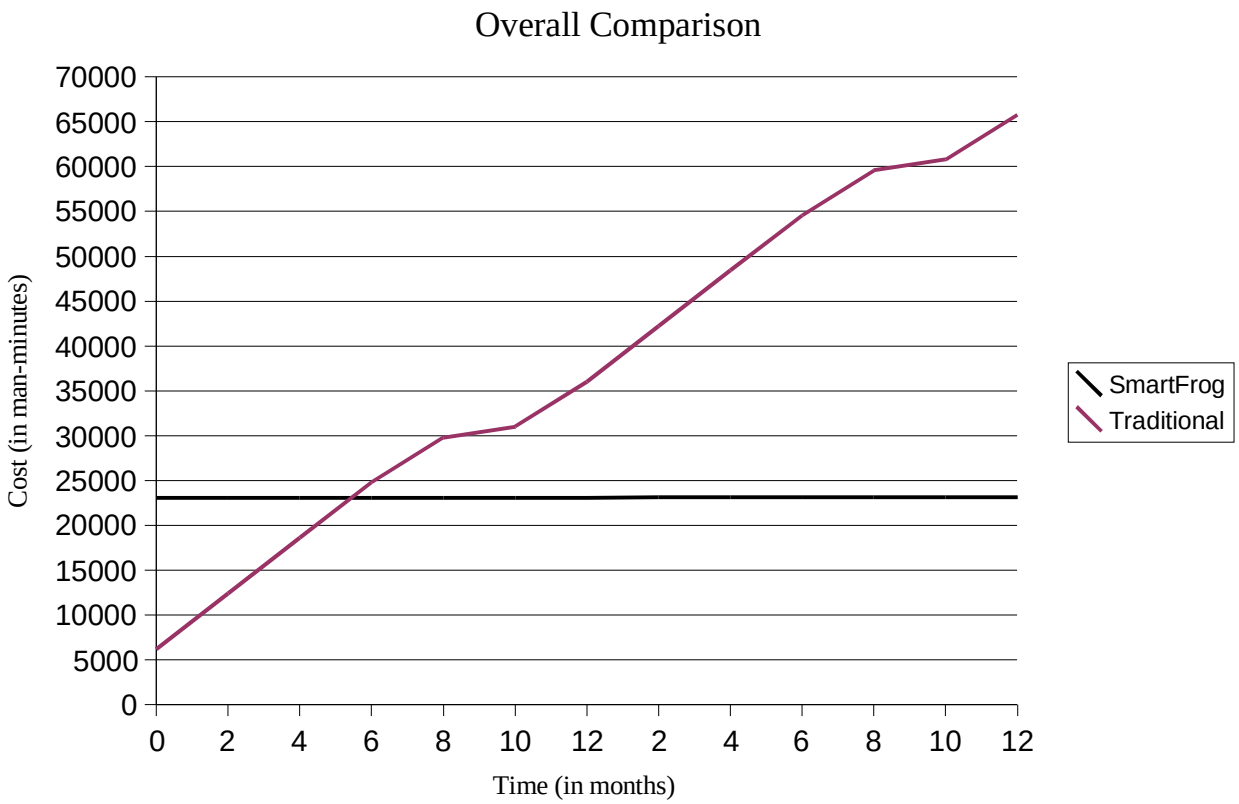

Figure 18

The traditional approach has a much lower starting cost in man-minutes because all the work that is needed is simply to deploy the infrastructure and the grid services on the computer. On the contrary, SmartFrog needs 23048 minutes to learn it and implement its components.

However, after that, SmartFrog's line is almost kept completely horizontal since the updates are easy and fast, whereas the traditional approach has a line, which rockets high

up. After two year the total cost in man minutes is almost triple than what it is for SmartFrog.

The table used for the chart in Figure 18 can be seen in Figure 19.

Months	SmartFrog	Traditional
0	23048	6200
2	23058	12400
4	23068	18600
6	23078	24800
8	23080	29800
10	23088	31000
12	23090	36000
2	23100	42200
4	23110	48400
6	23120	54600
8	23122	59600
10	23130	60800
12	23132	65800

Figure 19

I should note again that the scenario upon which the above chart is based, is arbitrary. We assumed what the number of computers should be, the number of grid services and the number of upgrades. But the length of each upgrade, for infrastructure and grid services is based on our calculations and should therefore be considered relatively accurate. The scenario should give a good idea about what the administrator stands to gain from using SmartFrog.

10 FURTHER WORK AND ENHANCEMENTS

10.1 SmartFrog

The development of a final executable prototype consumed significant amount of the project time. The functional capabilities incorporated into the design and final implementation constitutes the minimum operational functionality to make the prototype a “prove of concept”. The development efforts focused on the core requirements of the project: building a Grid Service deployment system using SmartFrog as a base platform. Due to time constraints and limited expertise and manpower the final executable system's characteristics have not been explored to the full extend of its capabilities, the capabilities of the underlying framework or to incorporate any other suitable software solutions. The following paragraphs cover some of the major issues or potential directions that eventually require more research and development effort.

As it stands right now the execution sequence, the system goes through, in run-time relies on status messages indicating the success or failure of a given component or module in that sequence. The status messages in turn rely exclusively on the default functionality of the SmartFrog-provided *Try* component. There is an unresolved issue occurring with the use of combination of components handling Unix OS command and shell script executions. SmartFrog system seems to accept as successful the completion of the deployment of the components only, regardless of the exit status of the executed OS command or shell file. Very often a failure exit status does not propagate up the component hierarchy resulting in the mention *Try* components incorrectly indicating a successful execution that further affects the dependent sequence logic.

The management abilities provided to the user are directly connected to that status message passing mechanism mentioned in the previous paragraph. Considering that all underlying SmartFrog components are implemented in Java, the management functionality of the prototype should be significantly improved by further exploring the Sun's Java Management Extensions (JMX) infrastructure. Moreover, the SmartFrog documentation indicates that the platform incorporates the JMX technology.

The submitted prototype has been tested on local-area settings in UCL Computer Science department laboratories. The deployed Grid Services technology is an extension of the Web Services standards. Inherently they are built to be used over wide-area/Internet

settings. A logical next development step would be expanding and testing the prototype performance when used across organizational boundaries. A significant attention should be paid to issues with computer security and the computer security policies put in place by potential users' organizations.

As part of the final prototype version there is a module running on each deployment host checking for available resources and presence of any partial or full deployment system infrastructure. The functionality of this module, however, is only rudimentary consisting of periodic resource checking and reporting. The module could be expanded to incorporate the more complex capabilities of making the deployment system adapt dynamically to changes in the hosting environment. A good example of such functionality is demonstrated by an example solution developed by the creators of SmartFrog and provided to the public trough the SmartFrog's web site – “Dynamic Web Server Demonstrator”.

Because of the restrictions mentioned in the first paragraph the implementation of the submitted deployment system was targeted for the Unix type OSs. The system was tested to successfully run on Sun's Solaris 8 and RedHat Linux 9. To take advantage of the fact that the underlying SmartFrog platform is implement in Java, thus, capable of running on different types of OSs, further effort should be made towards making the executable solution be more “aware” of the little differences among the different Unix OSs as well as be portable to MS Windows OS.

10.2 Security

In the context of this project, the issue of security spans across two distinct levels; the grid services level and the level of the application that carries out the grid service deployment.

The first level is concerned with the configuration of security for the grid platform; the Globus Toolkit. The goal is to enable grid services to use GT3’s security infrastructure. After conducting research we determined that theoretically GT3’s security configuration is possible, if not quite easy, to achieve and that it can be performed as part of the Smartfrog deployment steps. Our assumptions were that the GT3 simpleCA will be used, which is sufficient for simple grid services, and that the hosts, which the grid services are deployed, are going to request certificates from the CA that resides to a central machine. The process involves transferring the CA's distribution package, which was created

during the CA's installation on the central machine [25], and modifying a particular script. However, we decided that not deal with the configuration of the GT3.2 security as the project is concerned merely with the deployment of grid services and in particular a grid application that do not implement security.

The second level is concerned with the secure deployment of grid services and the security infrastructure SmartFrog [16] [19] uses itself. Security in this case is a key issue of concern, and specifically its aspects concerning privacy, integrity and authentication. Smartfrog could be used as a platform to deploy malicious code, such as viruses or Trojan horses. Therefore, strong authentication mechanisms should be employed to combat this problem. Moreover, the communications between the Smartfrog daemons may take place over insecure networks or the Internet. An attacker may eavesdrop on the exchange of system description files and deployment configurations, and obtain sensitive configuration information or even modify it. Therefore, privacy and/or integrity should be ensured. Finally, SmartFrog allows resources such as configuration descriptions, java bytecode, scripts and executable files to be downloaded from web servers. This imposes security risks in the light of loopholes found on web servers very often. Therefore, there is a need to ensure the integrity of the resources being downloaded.

“Two important steps must be carried out to ensure that the security of the SmartFrog framework is not compromised. The first is to initialize the SmartFrog security infrastructure and the second is to ensure that every resource (test file, URL, etc) is loaded through the secure mechanisms provided” [19]. Our research showed that the first step can be achieved by running Ant scripts provided by the SmartFrog distribution and then configuring the infrastructure appropriately. The second step can be carried out by signing the jar files that contain the resources so that the two SmartFrog features of remote class loading and downloading from web servers are secured. Moreover, inter-domain deployment with the use of multiple CAs is another issue, which falls under the umbrella of the security. All these are areas for further investigation.

11 CONCLUSION

The primary goal of this project has been the investigation of the possibility of using a deployment platform for distributed systems, such as Hewlett- Packard's Smart Framework for Object Groups - SmartFrog, to automatically and easily deploy Grid Services over a large number of hosting environments. The process involved not only theoretical research into the availability of various deployment applications and frameworks and their capabilities but also significant amount of development effort for the design and implementation of the executable prototype of a proposed solution.

During the theoretical research phase of the project a number of software configuration and deployment platforms provided by different vendors were explored. These included SoftwareDock by a group of researchers from the Software Engineering Research Laboratory at the University of Colorado, Tivoli Configuration Manager by IBM, JMX (Java Management Extensions) by Sun Microsystems and the above mentioned SmartFrog. They were carefully analyzed not only for the operational capabilities they offered but also to their ease of use, implementation, learning curve and suitability to the particulars of the project requirements.

The executable prototype of a proposed solution for deployment of Grid Services is entirely based on SmartFrog framework. This framework is capable of scaling from small systems to very large. The development of the prototype took advantage of the ability, SmartFrog provides to the platform user, to alter the low-level semantics by replacing functional units while still providing standard capabilities by offering default implementation of those units. The final working system has been successfully tested and it has proved to:

- operate in a variety of environments, ranging from a single machine to local-area distributed setting that leverages the connectivity offered by organizational networks;
- provide a way to access and reason about a software system configuration, which includes dependencies and constraints inherent in the system;
- make it possible to monitor the environment surrounding a deployed system, watch for changes in that environment;
- provide a way to describe site and software system configuration information, which includes the hardware and software environment at a site.

It has to be noted, though, that the above mentioned characteristics have not been developed to the full extent of the underlying platform and the prototype's capabilities. Some suggestions for further work in areas of potential interest have been outlined in the Further Work section. However, all the laboratory tests as well as the measures taken have clearly demonstrated that deployment systems are more quickly implemented using the technology and the structure imposed upon the implementations by the use of SmartFrog should be beneficial for long-term reliability, usability and manageability of the said dependent deployment systems.

APPENDIX A: BIBLIOGRAPHY

[1] A.Carzaniga, A. Fuggetta, R.S. Hall, A. van der Hoek, D. Heimbigner, A.L. Wolf. (1998). “A Characterization Framework for Software Deployment Technologies”. Dept. of Computer Science, University of Colorado.
http://www.cs.colorado.edu/department/publications/reports/docs/CU-CS-857-98.pdf

[2] Goonatilake, C., (2004). Comparing Grid Technologies for the Evaluation of Computationally Demanding Financial Applications. University College London.

[3] Hall, R.S., et al., (1997). “The Software Dock: A Distributed, Agent-based Software Deployment System”. Dept. of Computer Science, Univ. of Colorado
http://www.cs.colorado.edu/department/publications/reports/docs/CU-CS-832-97.pdf

[5] Foster, I., Kesselman, C. and Tuecke, S. (2001). “The Anatomy of the Grid: Enabling Scalable Virtual Organizations”. www.globus.org/research/papers/anatomy.pdf.

[6] CVS: Concurrent Versions System. https://www.cvshome.org/

[7] Eclipse. http://www.eclipse.org/

[8] Foster, I., Kesselman, C., eds. (2003). “The Grid: Blueprint for a New Computing Infrastructure”. Morgan Kaufmann.

[9] Foster, I., Berry, D., Djaoui, A., Grimshaw, A., Horn, B., Kishimoto, H., Maciel, F., Savva, A., Siebenlist, F., Subramaniam, R., Treadwell, J., Reich, J.(2004)."The Open Grid Services Architecture, Version 1.0". Open Grid Services Architecture Working Group.
https://forge.gridforum.org/projects/ogsa-wg/document/draft-ggf-ogsa-spec/en/19

[10] Banks, T., (2004). Open Grid Service Infrastructure Primer. OGSI-WG,
https://forge.gridforum.org/docman2/ViewCategory.php?group_id=43&category_id=392

[12] Foster, I., Kesselman, C., Nick, J., Tuecke, S. (2002). “Grid Services for Distributed System Integration”.
http://www.globus.org/research/papers/ieee-cs-2.pdf

[13] Foster, I. Kesselman, C. Nick, J., Tuecke S. (2002). “The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration”. The Globus Project
http://www.globus.org/research/papers/ogsa.pdf.

[15] Berman, F., Fox, G., Hey, T., (2003). Grid computing : making the global infrastructure a Reality. John Wiley and Sons Ltd.

[16] Goldsack, P., Guijarro, J., Lain, A., Mecheneau, G., Murray, P., Toft. P.(2003). "SmartFrog: Configuration and Automatic Ignition of Distributed Applications". HP Labs, Bristol, UK
http://www.hpl.hp.com/research/smartfrog/papers/SmartFrog_Overview_HPOVA03.May.pdf
[17] Tuecke, S., Czajkowski, K., Foster, I., Frey, J., Graham, S., Kesselman, C., Maquire, T. Sandholm, T., Snelling, D., Vanderbilt, P.(2003)."Open Grid Services Infrastructure (OGSI), Version 1.0".
https://forge.gridforum.org/docman2/ViewCategory.php?group_id=43&category_id=392

[18] Chappell, D., Jewell, T., (2002). Java Web Services. O'Reilly

[19] "The SmartFrog Reference Manual".(2004).HP Labs, Bristol, UK
http://www.hpl.hp.com/research/smartfrog/papers/sfReference.pdf

[20] The Apache Ant Project http://ant.apache.org/

[21] Armstrong, E., Ball, J., Bodoff, S., Carson, D., Fordin, S., Green, D., Evans, I., Haase, K., Jendrock, E., (). The Java Web Services Tutorial. Sun,
http://java.sun.com/webservices/docs/1.3/tutorial/doc/index.html

[22] "SmartFrog WorkFlow".(2004).HP Labs, Bristol, UK
http://www.hpl.hp.com/research/smartfrog/papers/sfWorkflow.pdf

[23] Apache Jakarta Tomcat. http://jakarta.apache.org/tomcat/

[25] Sotomayor, B., (2004). The Globus Toolkit 3 Programmer's Tutorial.
http://www.casa-sotomayor.net/gt3-tutorial-unstable/multiplehtml/ch12s03.html

[26] http://www.cs.ucl.ac.uk/staff/W.Emmerich/lectures/3C05-03-04/USDP.pdf