A Middleware Framework for Secure Mobile Grid Services

WONG, Sze Wing

A Thesis Submitted in Partial Fulfilment
of the Requirements for the Degree of
Master of Philosophy
in
Computer Science and Engineering

Abstract

A Grid is a set of resources distributed over wide-area networks that can support large-scale distributed applications. Regarding to current grid technologies, service providers can only provide stationary services which have no mobility during execution. To complement the deficiency of static Grid Services, the concept of Mobile Grid Services is proposed. Mobile Grid Services, the extension of the original static Grid services, are characterized by the ability of moving from nodes to nodes during execution. We can coordinate both services and resources by this kind of service mobility and maximize the resource usages of grids. This research aims to develop a middleware framework that supports Mobile Grid Services in a secure manner. By combining an existing mobile agent system (JADE) and a generic grid system toolkit (Globus), the Mobile Grid Service framework is constructed. The Mobile Grid Services are realized as Globus grid services with JADE mobile agent support. To support service development in the Mobile Grid Services middleware framework, an application programming interface called MGS API is implemented. The API consisting of the AgentManager class, Task Agent template, configurable Monitor Agent and Resource Information Service is used to provide both an easy and flexible environment for Mobile Grid Services development. General security mechanisms including authentication, authorization, agent permission, message integrity and confidentiality are provided in the framework. To protect service agents from malicious hosts, an agent protection mechanism employing execution tracing is particularly added. In this thesis, the details of the Mobile Grid Service framework as well as the MGS API with the security support will be presented. Some experiments are conducted to examine the performance of the Mobile Grid Services. Experimental results show that Mobile Grid Services can relieve the overloading problem with reasonable overall overheads.

論文摘要

網格是一群分佈在廣域網絡的資源，它可以支援大規模分佈式應用程序。就當前網格技術而論，服務供應商只能提供在執行中沒有流動性的固定式服務。「流動式網格服務」的概念爲補充靜態網格服務的不足而被提議出來。「流動網格服務」是原來的靜態網格服務之引伸，它的特點是能夠在執行中在節點之間移動。我們可以使用這種服務協調網格的服務和資源，達至最大限度的資源使用。這項研究的目的是建立一個能夠以安全方式支援「流動式網格服務」的中介軟體架構。「流動式網格服務架構」的構建是通過結合一個現有的行動代理程式系統(JADE)和一個通用的網格系統工具庫(Globus)。「流動式網格服務」實現爲一種加上行動代理程式支援的 Globus 網格服務。一個名爲 MGS API 的應用程式編寫介面爲支援「流動式網格服務架構」上的服務開發而被編寫出來。這個應用程式編寫介面包括「代理程式總管」、「任務代理程式模板」、「可更改配置的監控代理程式」及「資源信息服務」，爲「流動式網格服務」的開發提供了一個方便和靈活的環境。此架構提供一般的安全機制包括認證、授權、代理許可、信息完整性和保密規定。爲保障服務內的代理程式免受惡意主機攻擊，一個採用執行追蹤的代理程式保護機制特別加添到此架構之中。「流動式網格服務架構」以及擁有安全性支援的 MGS API 介面的細節將在這份論文中介紹。透過一些爲評估「流動式網格服務」的性能而進行的實驗，實驗結果表示「流動式網格服務」可以減輕超載問題，而整體費用是可以接受的。

Acknowledgements

I would like to thank my research supervisor, Professor NG Kam-Wing, who has provided numerous valuable guidelines and suggestions to me throughout my master of philosophy programme. The work described in this paper was partially supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region (Project no. CUHK4220/04E).

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Grid computing has been drawing a lot of attentions from both academia and industry in recent years. A Grid [1] is a set of resources distributed over wide-area networks that can support large-scale distributed applications. The Grid problem [2] is defined as the coordinated resource sharing and problem solving in dynamic, multi-institutional, virtual organizations. Grid computing enables users and systems to dynamically share resources, balance the loading on resources, and perform possible parallel processing.

Regarding to current grid technologies, the services provided by Grid Services providers are usually stationary. They are fixed on the Grid node machines providing the services and have no mobility. This means that the services cannot be moved to other nodes even if these other nodes have lots of idle resources. The idle resources will be wasted and the Grid resources cannot be utilized effectively. This static type of services also results in many drawbacks such as continuous connection and overload problem when a large amount of service requests are present at the same time.

Although the resource usage in a Grid may be increased by employing common load balancing mechanism to migrate Grid Services to idle hosts, the arrangement of service executing locations can be made only before services start. If a static service is forced to execute in another host, it will need to restart completely where the partially finished computation is wasted. Without service mobility, Grid Services with long execution times still cannot use the resources effectively in a dynamic Grid environment.

If the grid services can become mobile such that they can be moved to more appropriate nodes during their executions, grids can maximize their resource usages by coordinating both services and resources. Once Grid Services gain runtime mobility, load balancing is no longer limited to apply in the beginning stage of services. It can be carried out all the time according to the current Grid status. The service migration due to the load balancing is done without a full restart and thus the idle Grid resources can be used more efficiently. The ability of migration can also improve the practicability and flexibility of Grid Services.

By improving the original static Grid Service with the ability of migration, Mobile Grid Services can be realized such that the deficiency of static services is overcome. Our research aim is to develop a middleware framework to support Mobile Grid Services in a secure manner.

1.1 Contributions of this thesis

The main contribution of this thesis is proposing a middleware framework which supports a new type of Grid Services, called Mobile Grid Services in a secure manner. Mobile Grid Services is realized as the extension of the original static Grid Services by deploying mobile agent technology for providing mobility. The essential advantage is the ability of performing runtime service migration according to the dynamics of the Grid environment. To support the Mobile Grid Services, a Mobile Grid Service Framework is developed by combining an existing mobile agent system (JADE) and a generic grid system toolkit (Globus). Moreover, an application programming interface called MGS API is developed to ease the service development. The API provides the essential components in the Mobile Grid Service Framework and facilities to assist service developers performing programming and configuration works for the services. Furthermore, general security mechanisms such as authentication, authorization, message integrity and confidentiality are deployed into the framework to provide security support for the services. Hence, Mobile Grid Services can execute in a secure manner and be used for developing applications concerned with security issues. Finally, the framework gains the support of agent protection by integrating a security mechanism, called "Execution Tracing with Randomly-Selected Hosts" which protects an agent from attack by malicious hosts. Under the agent protection, any modification of code or data on a service agent is detected by the execution tracing.

1.2 Thesis structure

The following list outlines the structure of this thesis:

Chapter 2 Background The second chapter reviews some technologies used for developing the framework supporting Mobile Grid Services in this research.

Chapter 3 Research Issues in Mobile Grid Services The third chapter proposes the Mobile Grid Services which improve on the original static Grid services by adding the mobile ability of moving from nodes to nodes during execution. Besides, three main research issues including service migration, service sharing and discovery, and security in Mobile Grid Services are discussed.

Chapter 4 Mobile Grid Service Framework The fourth chapter proposes a middleware framework that supports Mobile Grid Services. This chapter includes the architecture, the components and the service execution's scenario of the framework.

Chapter 5 MGS API The fifth chapter introduces an application programming interface called MGS API. The API aims to provide both an easy and flexible environment for service development in the Mobile Grid Service framework.

Chapter 6 Security Support for Mobile Grid Services The sixth chapter presents the details of security measures (authentication, authorization, agent permission, message integrity and confidentiality) in the framework. The security facilities provided in the MGS API are also illustrated.

Chapter 7 Agent protection for Mobile Grid Service The seventh chapter introduces the agent protection in the Mobile Grid Service Framework. The implementation details of the mechanism as well as its strength and weakness are discussed.

Chapter 8 Performance Evaluation The eighth chapter presents the performance evaluation of the Mobile Grid Services. Some experiments are conducted for this purpose. Their results, analysis and the overheads estimation are discussed.

Chapter 9 Conclusion and Future Works The last chapter concludes the thesis and presents the future works.

Appendix A Administrator Guide for MGS API This appendix shows a full installation of the MGS API and the setup of the MGS platform.

Appendix B Developer Guide for MGS API This appendix presents the method of implementing and configuring a Mobile Grid Service, the useful tools provided and the interface of the MGS API.

Chapter 2

Background

In this chapter, the major technologies related to this research are reviewed. They are Web Services, Grid computing, Globus toolkit, mobile agent and Java Agent Development Framework (JADE).

2.1 Web Services

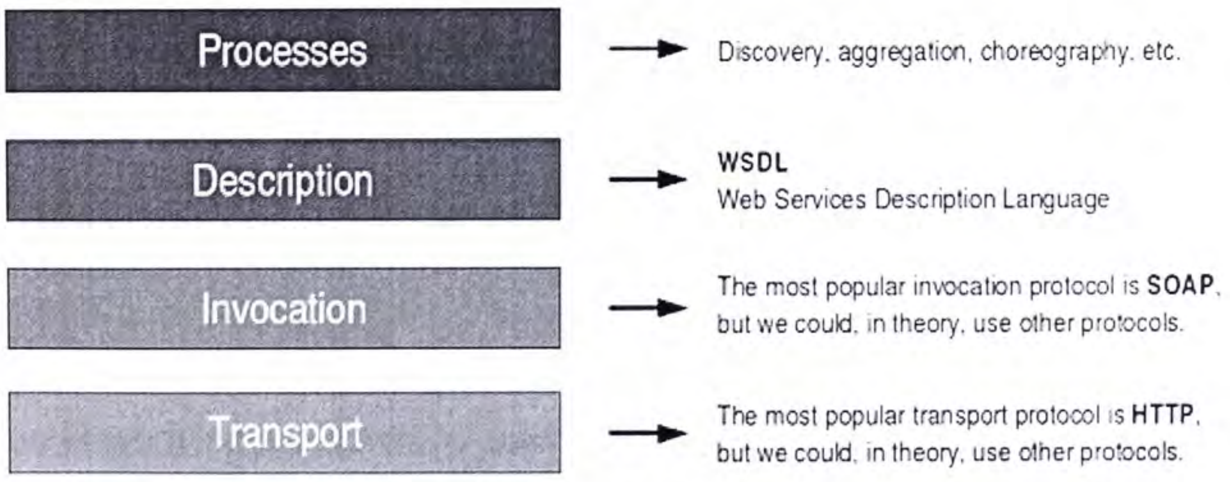

Processes
Discovery, aggregation, choreography, etc.
Description
WSDL
Web Services Description Language
Invocation
The most popular invocation protocol is SOAP, but we could, in theory, use other protocols.
Transport
The most popular transport protocol is HTTP, but we could, in theory, use other protocols.

Figure 2-1: The architecture of Web Services

Web Services is an emerging distributed computing paradigm focusing on simple, Internet-based standards (e.g., Extensible Markup Language: XML [3]) to address heterogeneous distributed computing. Web services standards are being defined within the W3C. Web services define a technique for describing software components to be accessed, methods for accessing these components, and discovery methods that enable the identification of relevant service providers. They are both platform-independent and language-independent.

A Web Service consists of the following four components (Figure 2-1):

Service Process The process of the Web Service, it usually involves more than one service. For example, the discovery process will gather resource information from different Web services

Service Description The Web Services are self-descriptive, meaning that once we have located a Web service, we can ask it to 'describe itself' and tell us what operations it supports and how to invoke it. This is handled by the Web Services Description Language (WSDL) [4], which is used to achieve self-describing, discoverable services and interoperable protocols, with extensions to support multiple coordinated interfaces and change management

Service invocation Invoking a Web Service involves passing messages between the client and the server. Simple Object Access Protocol (SOAP) [5] specifies how we should format requests to the server, and how the server should format its responses. In theory, we could use other service invocation languages (such as XML-RPC, or even some ad hoc XML language). However, SOAP is by far the most popular choice for Web Services.

Transport All these messages must be transmitted somehow between the server and the client. The protocol of choice for this part of the architecture is usually HTTP (Hypertext Transfer Protocol).

2.2 Grid Computing

A Grid [1] is a set of resources distributed over wide-area networks that can support large-scale distributed applications. Grid computing enables the virtualization of distributed computing and data resources such as processing, network bandwidth and storage capacity to create a single system image, granting users and applications seamless access to vast IT capabilities. Just as an Internet user views a unified instance of content via the Web, a grid user essentially sees a single, large virtual computer. With grid computing, organizations can optimize computing and data resources, pool them for large capacity workloads, share them across networks and enable collaboration.

In [2], the problem underlying the Grid concept is defined as the coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations. The sharing is not primarily file exchange but rather direct access to computers, software, data, and other resources. This sharing is highly controlled and defined clearly by resource providers and consumers with the details about what is shared, who is allowed to share, and the conditions under which sharing occurs. A set of individuals and/or institutions defined by such sharing rules form a virtual organization (VO).

2.2.1 Open Grid Services Architecture (OGSA)

OGSA [6] is a standard developed by the Global Grid Forum [7], and aims to define a common, standard, and open architecture for grid-based applications. The goal of OGSA is to standardize practically all the services one commonly finds in a grid application (job management services, resource management services, security services, etc.) by specifying a set of standard interfaces for these services.

OGSA defines uniform exposed service semantics (the Grid Service). Besides, it defines standard mechanisms for creating, naming, and discovering transient grid service instances, provides location transparency and multiple protocol bindings for service instances, and supports integration with underlying native platform facilities. Some sort of distributed middleware is needed to support the architecture. OGSA has chosen Web Services as the underlying technology of the distributed middleware in order to profit from the existing capabilities of Web Services.

2.2.2 Grid Services

For providing standard semantics for service interactions, OGSA defines Grid Service which is a Web Service that provides a set of well-defined interfaces and that follows specific conventions. The interfaces address discovery, dynamic service creation, lifetime management, notification, and manageability; the conventions address naming and upgradeability. This core set of consistent interfaces, from which all Grid services are implemented, facilitates the construction of higher-order services that can be treated in a uniform way across layers of abstraction.

Grid Services are characterized (typed) by the capabilities that they offer. A Grid service implements one or more interfaces, where each interface defines a set of operations that are invoked by exchanging a defined sequence of messages. The complete interface of a Grid service is described in a WSDL document and advertised through public registries.

2.3 Globus Toolkit

The Globus Toolkit [8] is an open source software toolkit, developed by the Globus Alliance [9], which we can use to create a Grid system. It is a realization of the OGSA [6] requirements. The toolkit includes high-level services that we can use to build Grid applications. It is organized as a collection of loosely coupled components. These components consist of services, programming libraries and development tools designed for building Grid-based applications. They are packaged as a set of components that can be used either independently or together to develop applications. The Globus Toolkit offers a development environment for producing new Grid services that follow the OGSA architectural principles.

The toolkit also includes a complete implementation of the Web Services Resource Framework (WSRF) specifications [10]. WSRF specifies stateful Web Services which is required by OGSA. It is a set of Web services specifications that defines conventions for managing "state" in the Web services context so that applications can reliably share changing information.

2.3.1 Components of Globus Toolkit 4

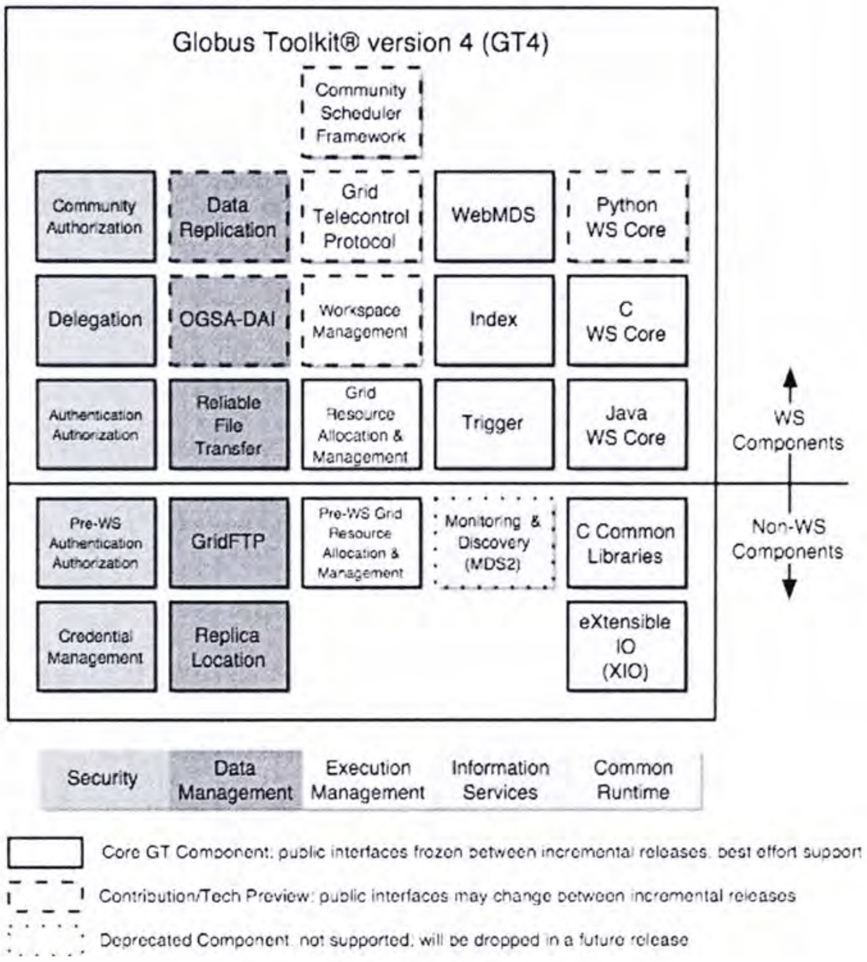

Globus Toolkit® version 4 (GT4)
Community Scheduler Framework
Community Authorization
Data Replication
Grid Telecontrol Protocol
WebMDS
Python WS Core
Delegation
OGSA-DAI
Workspace Management
Index
C WS Core
Authentication Authorization
Reliable File Transfer
Grid Resource Allocation & Management
Trigger
Java WS Core
WS Components
Pre-WS Authentication Authorization
GridFTP
Pre-WS Grid Resource Allocation & Management
Monitoring & Discovery (MDS2)
C Common Libraries
Non-WS Components
Credential Management
Replica Location
eXtensible IO (XIO)
Security
Data Management
Execution Management
Information Services
Common Runtime
Core GT Component: public interfaces frozen between incremental releases; best effort support
Contribution/Tech Preview: public interfaces may change between incremental releases
Deprecated Component: not supported; will be dropped in a future release

Figure 2-2: GT4 Components

The latest version of the Globus Toolkit is Globus Toolkit 4 (GT4). Fig. 2-2 illustrates the components provided by the GT4 Toolkit. The five families of the components are:

Common Runtime It provides a set of fundamental libraries and tools which are needed to build both WS and non-WS services. Libraries in different programming languages including C, python and Java are supplied for the implementation of services.

Data Management It allows us to locate, manage and transfer data on the grid.

Three primary data management tools are the GridFTP for memory-to-memory as well as disk-to-disk data access in the grid, the Reliable File Transfer (RFT) service for managing multiple transfers and the Replica Location Service (RLS) for maintaining location information for replicated files.

Information Service This component is also referred to as the Monitoring and Discovery Service (MDS) which provides a suite of Web services to monitor and discover resources and services on Grids. It provides query and subscription interfaces to arbitrarily detailed resource data and a trigger interface that can be configured to take action when pre-configured trouble conditions are met.

Execution Management This component deals with the initiation, monitoring, management, scheduling and coordination of executable programs (jobs) in a Grid. GT4 supports the Grid Resource Allocation and Management (GRAM) interface as a basic mechanism for these purposes.

Security GT4 provides security tools concerned with establishing the identity of users or services (authentication), protecting communications, and determining who is allowed to perform what actions (authorization), as well as with supporting functions such as managing user credentials and maintaining group membership information.

2.3.2 Grid Security Infrastructure (GSI)

The Grid Security Infrastructure (GSI) [11] is the basis of the GT4's security layer. The GSI protocol provides single sign-on, credential delegation, authentication through X.509 certificates [12], communication protection, and several authorization schemes. In brief, single sign-on allows a user to authenticate once and thus create a proxy credential that a program can use to authenticate with any remote service on the user's behalf. Delegation allows for the creation and communication to a remote service of delegated proxy credentials that the remote service can use to act on the user's behalf, perhaps with various restrictions; this capability is important for nested operations.

2.4 Mobile Agent

The agent paradigm applies concepts from artificial intelligence and speech act theory to the distributed object technology. The paradigm is based on the agent abstraction. Generally, agents are software entities with the following attributes:

- autonomous

  Agents have a degree of control on their own actions, they own their thread of control and, under some circumstances, they are also able to make decisions;
- proactive

  Agents do not only react in response to external events (i.e. remote method call) but they also exhibit a goal-directed behavior and are able to take initiative;

- social

  Agents are able to interact with other agents in order to accomplish their task and achieve the complete goal of the system.

Mobile agents are autonomous software processes that can move from node to node in a network to access services provided there and to communicate with other mobile agents. When an agent decides to migrate to another node, the agent's code, data and execution state are captured and transferred to the next node, where it is instantiated after arrival.

The advantages including customization, reduced communication bandwidth and asynchronous task execution make the mobile agent paradigm very attractive especially in systems where network bandwidth is low or network connection cost is high; and for applications where the remote data to be processed is vast or the task to be performed is time-consuming.

To standardize the mobile agent system, some standards are developed. One of them is organized by FIPA [13].

2.4.1 Foundation for Intelligent Physical Agents (FIPA)

FIPA [13] is a standards organization that promotes agent-based technology and the interoperability of its standards with other technologies. It aims to produce software standards specifications for heterogeneous and interacting agents and agent based systems.

The FIPA standard fully embraces the agent paradigm. In particular, it defines the reference model of an agent platform and a set of services (including Life cycle

Management, White page service, Yellow page service and Message Transport service) that should be provided. The collection of these services, and their standard interfaces, represents the normative rules that allow a society of agents to exist, operate, and be managed.

Another main asset of the FIPA standard is the Agent Communication Language (ACL). The FIPA ACL is based on the speech act theory and on the assumptions and requirements of the agents' paradigm. FIPA standardized an extensible library of 22 communicative acts that allow representation of different communicative intentions (such as requesting, proposing, informing, querying, calling for a proposal, refusing, etc.). FIPA also defined the structure of a message that allows to represent and convey information useful to identify the sender and receivers, the content of the message and its properties (e.g. the encodings and the representation language), and, in particular, information useful to identify and follow threads of conversation between agents and to represent timeouts for the communication. Common patterns of conversations, called interaction protocols, have been also defined by FIPA so that they provide agents with a library of patterns to achieve common tasks.

2.5 Java Agent Development Framework (JADE)

JADE [14] is an open source software development framework aimed at developing multi-agent systems and applications conforming to FIPA standards for intelligent agents [13]. JADE has been fully coded in Java and includes two main products: a

FIPA-compliant agent platform and a package to develop Java agents.

JADE includes the run-time environment that provides the basic services and that must be active on the device before agents can be executed. Each instance of the JADE run-time is called a container. The set of all containers is called a platform (can be distributed on several hosts) and provides a homogeneous layer that hides from agents (and to application developers also) the complexity and the diversity of the underlying information (hardware, operating systems, types of network, JVM).

The JADE Agent Platform complies with FIPA specifications and includes all those mandatory components that manage the platform. The components include the Agent Communication Channel (ACC), the Directory Facilitator (DF) and the Agent Management System (AMS). The ACC acts as the Message Transport System for controlling all the exchange of messages within the platform; the DF is responsible for providing the default yellow page service in the platform; the AMS exerts supervisory control over access to and use of the Agent Platform. It is responsible for white-page service, life-cycle service and maintaining a directory of agent identifiers (AID) and agent state. Each agent must register with an AMS in order to get a valid AID.

All agent communication is performed through message passing, where FIPA ACL is the language to represent messages. The communication architecture offers flexible and efficient messaging, where JADE creates and manages a queue of incoming ACL messages, private to each agent; agents can access their queue via a combination of several modes: blocking, polling, timeout and pattern matching based.

On the other hand, JADE includes the libraries (i.e. the Java classes) required to develop application agents. Basically, agents are implemented as one thread per agent, but agents often need to execute parallel tasks. Further to the multi-thread solution, offered directly by the JAVA language, JADE supports also scheduling of cooperative behaviours, where JADE schedules these tasks in a light and effective way. The run-time includes also some ready to use behaviours for the most common tasks in agent programming, such as FIPA interaction protocols, waking under a certain condition, and structuring complex tasks as aggregations of simpler ones.

Using JADE, application developers can build mobile agents, which are able to migrate or copy themselves across multiple network hosts. In this version of JADE, only intra-platform mobility is supported, that is a JADE mobile agent can navigate across different agent containers but it is confined to a single JADE platform.

The agent platform provides a Graphical User Interface (GUI) for the remote management, monitoring and controlling of the status of agents. Upon the core of JADE, a number of graphical tools have been implemented to supports the debugging phase.

2.5.1 JADE-S

JADE-S is an add-on of the JADE platform that provides support for security in multi agent systems such as Authentication and Authorization. It is based on the Java security model and extends it for multi-agent systems. JADE-S makes the JADE platform a controlled multi-user environment, where all the components are owned by authenticated users, whom in turn are authorized by the platform administrator to perform only certain privileged actions.

Chapter 3

Research Issues in Mobile Grid Services

In this chapter, extension of standard Grid Services - Mobile Grid Services are proposed and three main research issues in Mobile Grid Services are discussed. They are Service Migration (how to add mobility to Grid Services), Service Sharing and Discovery (how to publish and find services), and Security (how to protect services and resources). Some related research focusing on these issues will be considered.

3.1 Mobile Grid Services

Grid Services are "stateful" Web Services which conform to the OGSA [6] standard. To improve the Grid Services, Mobile Grid Services are proposed as an extension. Unlike the original static Grid Services, Mobile Grid Services are able to move from nodes to nodes in the Grid because of their mobility capability. They can leave their hosts and migrate to other Grid nodes containing more idle resources. We can coordinate both services and resources by this kind of service mobility and thus the resource usages of grids can be maximized. Mobile Grid Services can be seen as a special kind of Web Services with state and mobility.

Compared with the standard grid services, Mobile Grid Services are characterized by the service mobility. With the help of the mobility, Grid services can travel throughout the Grid to get information from Grid nodes, execute in those nodes and bring the results back to their original hosts. Imagine that a service requires a lot of resources for its execution. If the service is implemented as a standard grid service, it will be blocked when the hosting machine runs out of resources. However, if the service is implemented as a Mobile Grid Service, it can move the execution to another host with plenty of resources. Moreover, the overload problem can be relieved due to the service migration. Even if a large amount of service requests are present at the same time, the execution load can be distributed to other hosts and overloading on the hosting machine is avoided. It can help to balance the load on all grid nodes.

Besides the advantages above, Mobile Grid Services can also travel throughout the Grid intelligently according to resource needs with the help of the mobility. This improves the flexibility of Grid Services. If a standard grid service needs to access massive data in other nodes, continuous connections between Grid nodes are essential. However, this is less critical for Mobile Grid Services where they can move to the nodes storing the data and get data from the Grid nodes locally, carry out the execution in those nodes and finally bring the results back to their original hosts. This may also reduce the network traffic of the data transmission.

3.2 Service Migration

The most obvious question for Mobile Grid Services is how to add mobility to the Grid Service. This is equivalent to how to move a running application from one grid node to another and resume the execution in the target location. To achieve this goal, there are many different approaches. One of them is inserting check-pointing functions into the application code [15]. Another approach accomplishes the objective by integrating mobile agent technology into grid computing. It uses mobile agents to encapsulate the application for providing mobility [16, 17].

3.2.1 Using Mobile Agent with Weak Mobility

MobiGrid [17] is a project which focuses on a framework for mobile agents support within the InteGrade grid environment [18]. It aims to allow an efficient utilization of computational resources for time consuming applications. In the framework, mobile agents are used to encapsulate the user applications. Applications can leave their currently running machines and migrate to other idle or more powerful machines in the execution.

The migration ability of MobiGrid agents comes from its mobile agent system. MobiGrid makes use of Aglet [19] which provides resources for mobile agent creation, migration, cloning, security, synchronization and message exchange. However, one drawback of the Aglet system is that only weak migration is provided. That means only the objects states and variables are preserved, not the state of execution stacks. In order to prevent the already computed results from missing during migration, the developer of the application is responsible for saving the present state of the application by calling the provided checkPoint() method at the appropriate time.

This approach of encapsulating an application into a mobile agent with weak mobility can use small slices of the available computational time of personal workstations, migrating to another machine whenever the local user requests his machine, always preserving the processing already done. However, as application programmers have to take care of saving the present state of the application, the difficulty and the time required for implementation is increased. It is unfavorable for the development of Grid services

3.2.2 Using Mobile Agent with Strong Mobility

While You're Away (WYA) [16] is a distributed system that aggregates the computation power of individual computer systems. WYA introduces the notion of "Roaming Computations" - Java-based programs that move around the network utilizing the resources of idle workstations. By employing mobile agent technology, “Roaming Computations” are realized. Similar to the MobiGrid project, WYA applications are encapsulated in mobile agents. The main difference of them is that the mobile agent system used by WYA is able to provide strong mobility instead of weak mobility.

WYA is based on the NOMADS mobile agent system [20], which uses the Aroma Virtual Machine (VM) [21] to provide strong mobility for Java-based agents. Unlike weak mobility which requires that a mobile agent restarts execution after a move, strong mobility can capture and transfer the full execution state of the agent.

An agent can request a move operation at anytime without wasting any resources used in previous computations. NOMADS also supports forced mobility, where external events can forcibly move agents from one system to another. This makes the migration transparent to an agent when it needs to move to another less busy host.

The most important benefit of this approach is the complete transparency of migration to the agents. The agents together with the applications can freely migrate from one place to another without any loss of computed partial results. At the same time, no extra concern about the state capture is required by the programmer. The capturing of the agent's execution state is done automatically by the mobile agent system. This is the reason why strong mobility is superior to weak mobility. However, WYA uses a modified version of JVM which is not desirable, since such JVMs usually become obsolete compared to new versions of Java 2 (not following Sun Microsystems' standardization).

3.2.3 Using Snapshots

Fukuda [15] has proposed a mobile-agent-based middleware that benefits remote computer users who wish to mutually offer their desktop computing resource to other Internet group members while their computers are not being used. In the framework, a mobile agent is used to represent a client user for coordinating his/her job over a computational grid, but not to encapsulate the application.

After locating a computer to run a user job, the mobile agent uploads all files necessary for execution from its client to the target computer. When this computer becomes unavailable (e.g. CPU busy) during job execution, the monitoring mobile agent will move the corresponding application process to another available machine. To accomplish application migration without completely restarting, this action requires transferring the application code as well as the running state. The application's running state is saved in an execution snapshot which is produced by calling a check-pointing function in the application. A language preprocessor is needed to automatically insert check-pointing functions into the user application source code at compilation time. A back-up snapshot is periodically stored in several machines in the grid during job execution, so that a mobile agent can retrieve a suspended process from the latest snapshot when necessary.

One advantage of this approach is that programmers need not pay attention to the snapshot production. Besides, a computation can even be recovered from a sudden machine halt. Nevertheless, since the migration may not be predicted in advance, some of the computed results (after the latest snapshot) will still be lost. Moreover, if migration rarely occurs, the snapshots produced periodically will be a waste of the grid resources.

3.2.4 Summary

All three approaches provide migration ability to Grid Services (application) and prevent the application from completely restarting after the movement. Among these three approaches, encapsulating an application in a mobile agent with weak mobility seems to be the best solution to Mobile Grid Services. This approach takes the advantages of mobile agent's mobility and autonomy features. The Grid Services can move to more appropriate nodes by their own logic and mobility. They can work independently without the monitoring from others. Besides, no modified JVM is required to be installed throughout the grid for the execution of mobile agents. Therefore, it is suitable for supporting mobility in Mobile Grid Services.

3.3 Service Sharing and Discovery

In a Grid, various Grid services (including resources) are provided by service provider nodes and can be utilized by Grid users. To support users searching for their desired services, some means must be provided for publishing and discovering services. Service Sharing and Discovery is important for Mobile Grid Services to find appropriate nodes to migrate to and to employ other existing services. A simple method is using a centralized coordinator to store all the information about the grid [22]. To increase scalability, the grid may be divided into some clusters. Each cluster contains a manager to keep the update information of all grid nodes in the cluster [18]. Web Services protocols including WSDL [4] and UDDI [23] can also be used for registering and discovering Mobile Grid Services [24].

3.3.1 Centralized Model

Hulaas [22] outlines a Computational Grid deployment protocol which is entirely based on Java, leveraging the portability of this language for distributing customized computations throughout large-scale heterogeneous networks. In this computation model, an operator is responsible for maintaining the whole grid. All resource donators and clients should register at the operator. The information of all the services and resources in the grid is kept by this centralized operator and will be updated occasionally.

To deploy an application to the Grid, a client should send a deployment descriptor containing information such as resource requirements and QoS parameters to the operator. According to the deployment descriptor details and the grid information, the operator will choose an appropriate set of donators taking into consideration their current load and dispatch a deployment agent to a suitable place for coordination of the client application. The client application code and data are then moved to the donator's location, where the computation takes place.

This centralized approach is simple to design and manage. All Grid information is stored in a centralized control structure. Resources and services are published to the Grid by registering at the centralized manager while clients search resources or services by asking the manager. However, all load for the service sharing and discovery is concentrated at the manager. This leads to a bottleneck as well as the single point of failure problem. Moreover, such a centralized model is not scalable when the size of the grid grows.

3.3.2 Division into clusters

The MobiGrid project [17] is based on InteGrade [18] which is an object-oriented middleware Grid infrastructure. InteGrade grids are structured in clusters, each consisting of groups of computers. Clusters are then arranged in a hierarchy, allowing a single InteGrade grid to encompass potentially millions of machines. In each cluster, there is a Cluster Manager node which is responsible for managing that cluster and communicating with managers in other clusters. Service and Resource information within the cluster is also maintained in the Cluster Manager.

The Global Resource Manager (GRM) in the Cluster Manager periodically receives information about the node status such as CPU usage from the cluster nodes. When a grid user submits an application for execution, the GRM selects candidate nodes for execution, based on resource availability and application requirements. The GRM will use its local information about the cluster state as a hint for locating the best nodes to execute an application. If necessary, the discovery process can be done across clusters through their corresponding Cluster Managers.

This approach tries to increase scalability by grouping grid nodes into clusters. The Service sharing and discovery load of the whole grid is distributed to the Cluster Managers. Nevertheless, the bottleneck and single point of failure problem still exist in this model.

3.3.3 Using Web Services Protocols

The Open Grid Service Architecture (OGSA) [6] is a promising architecture standard for future grid architectures and aims to combine Web Services into the grid architecture. OGSA abstracts all resources to be Grid Services.

Web Services define a technique for describing software components to be accessed, methods for accessing these components, and discovery methods that enable the identification of relevant service providers. Two important components of the Web Services protocols are the Universal Description, Discovery, and Integration (UDDI) [23] and the Web Services Description Language (WSDL) [4]. UDDI is a standardized method for publishing and discovering information about web services while WSDL is an XML document for describing Web Services.

Similar to the OGSA approach, the work of Zhang [24] uses Web services to support the dynamic registering and discovery of services in heterogeneous environments. It proposes the concept of Grid Mobile Service (GMS) that is defined as an intelligent code service wandering in grid nodes to accomplish certain tasks and provide certain services. Besides, the critical factors and the implementation methods of GMS are also discussed. To adapt to the Mobile Service interface, GMS extends the WSDL protocol by adding new elements for describing Mobile Grid Services and follows the web services approach to publish, identify and explain mobile services. UDDI is used to register and discover services and resources. A user or an application can access an UDDI register in order to publish or search for a grid service.

This approach utilizes the Web Services standards for publishing and searching Grid Services. It gains the benefit of properly supporting heterogeneous nodes in the Grid environment as the Web Services Protocols are designed to support interoperability (i.e. independence of the transport protocols, programming languages, programming models and system software). It also adapts the current trend of the convergence of Web and Grid services promoted by OGSA

3.3.4 Summary

The first two approaches suffer from the bottleneck and the single point of failure problem and thus cannot be considered as an appropriate solution for service sharing and discovery of Mobile Grid Services. The third approach using Web Services Protocols is more preferable because of its ability of supporting interoperability over distributed heterogeneous environments. This precisely fits the Grid environment which usually contains heterogeneous nodes interconnected together. Moreover, this approach maintains the similarity of Mobile Grid Services and ordinary Grid Services. The conversion of existing Grid Services to Mobile Grid Services is simplified.

3.4 Security

As resources are distributed throughout the grid, any grid user is able to access them when there is not any security policy. The problem is significant for the grid with Mobile Grid Services which commonly exploit resources from various grid nodes. Therefore, some measures should be taken to protect them against being abused by malicious users or applications. Resource control and accounting [15, 16] can detect any improper resource utilization by malicious applications. This includes placing limits on the resources which can prevent unlimited abusing. To further improve the security, delegation documents [25] can be used to ensure that consumers of the resources should be trusted.

3.4.1 Resource control and accounting

In the Grid computing infrastructure of Hulaas *et al.* [22], the secure execution of the mobile code (customized computation) is supported by JavaGridKernel, a Java-based middleware developed for providing fully portable resource accounting and resource control. JavaGridKernel transforms application classes and libraries, including the Java Development Kit, in order to expose details concerning their resource consumption during code execution. The bytecode of Java classes is rewritten before they are loaded by the JVM. Currently, CPU, memory and network bandwidth control are addressed.

The resource control is used to prevent malicious or erroneous code from overusing the resources of the host where it has been deployed (e.g., denial-of-service attacks). Resource consumption information can be used to detect strange behaviors of malicious applications. Moreover, it enables the charging of clients for the consumption of their deployed applications.

In the WYA system [16], the NOMADS mobile agent system [20] is also responsible for providing the dynamic resource control mechanism. Various limits may be placed on the resources that can be consumed by user applications. These limits include both rate limits (e.g. percentage of CPU usage, disk and network read or write rates) and quantity limits (e.g. disk space used). These resource limits may be dynamically adjusted at a fine level of granularity and are enforced transparently to the application execution. This can be used to protect hosts against malicious or buggy agents as well as to prioritize the execution of agents. Consequently, resources will never be overused by any agent.

These security mechanisms focus on preventing grid resources from being abused by the user applications. Resource accounting allows the system to measure and keep track of the resources consumed by an application on a host. Resource abuse can be detected by analyzing these data. Resource control can protect resource providers and prevent unlimited abuse of their resources. However, applying resource control and accounting as the only security measures for Mobile Grid Services are clearly insufficient. These mechanisms focus only on the utilization of resources. The trust of the application is not considered. Execution of malicious applications using apparently normal amount of resources will not be prohibited or detected.

3.4.2 Using delegation document

An instance-oriented security mechanism [25] is proposed to deal with security threats in building a general-purpose mobile agent middleware in a Grid environment. The proposed solution imports security instance, which is an encapsulation of one set of authorizations and their validity specifications with respect to the agent's specific code segments, or even the states and requests. Users and applications can define several kinds of security instances and their possible operations, according to the application's own logics. The instances should be signed by its creator. In the instance-oriented security framework, an agent representing the client carries a delegation document containing instance details and goes to the target host for application execution. By checking the signature and the details of the instance, resource providers can ensure that their resources are used by applications from trusted parties in an acceptable manner.

Once a delegation document migration is carried out between hosts, one or more handover operations must be performed. The maximum number of times of performing the handover operation is specified in the instances. When the instance reaches the max handover times, it will be regarded as invalid. This mechanism is used to handle the case that suddenly malicious hosts (originally trusted) abuse the delegation document and prevent unlimited diffusion of the potential damage.

By using the delegation document, resource providers can follow their own logics to allow only those applications fulfilling their requirements to execute. The utilization of resources can be kept under control and the code executed can also be assured to be trusted. Drawbacks of this method include the extra computation required for handling the delegation documents, network bandwidth required for the transfer of the delegation documents, and the extra effort for the developers.

3.4.3 Summary

Both approaches are able to protect Grid resources against misuse by malicious applications or Grid Services. Resource abuse can be detected or prevented. The resource control and accounting approach, which has no extra overheads as those of using delegation document, is better for a LAN environment where all hosts and users are supposed to be trusted. However, for the Grid environment which is a loose coupling scenario for large numbers of Virtual Organizations (VO) over the Internet, malicious users are more likely to be present and this approach seems to be insufficient as a result of its lack of trust checking. Therefore, the approach of using delegation document is preferable for Mobile Grid Services.

Chapter 4

Mobile Grid Service Framework

This chapter proposes a middleware framework that supports Mobile Grid Services in a secure manner. The framework is constructed by combining an existing mobile agent system (JADE) [14] and a generic grid system toolkit (Globus) [8]. The Mobile Grid Services are realized as Globus grid services with JADE mobile agent support. In this chapter, the details of the proposed framework including the architecture, the components and the service execution's scenario are presented.

4.1 Proposed Framework Overview

The main concern of this research is providing a middleware framework for Secure Mobile Grid Services. The framework is constructed by joining the Java Agent Development Framework (JADE) [14] and the Globus Toolkit [8].

In our framework, a JADE mobile agent (supporting weak mobility) is used to encapsulate the actual working unit of the application task. JADE containers are setup throughout the grid to provide platforms for mobile agents to move on. The communication between these JADE agents is accomplished by exchanging messages in the format of the Agent Communication Language (ACL) defined by the FIPA specification [13].

For each application, a Globus grid service will create the corresponding mobile agents and act as a relay between users and the mobile agents. In this way, Mobile Grid Services are realized as a type of grid services which distribute the actual working tasks to mobile agents. The advantage of this design is that the existence of the relay part makes the Mobile Grid Services conform to the Globus grid services architecture. At the same time, the mobile agent part is able to exploit useful resources in other grid nodes.

The three main concerns (Service Migration, Service Sharing and Discovery, and Security) are handled in the framework as follows:

4.1.1 Service Migration

Similar to the approach using mobile agents with weak mobility [17], a JADE mobile agent (supporting weak mobility) is used to encapsulate the actual working unit of the application task in this framework. JADE containers are setup throughout the grid to provide platforms for mobile agents to move on. For each application, a Globus grid service will act as a relay between users and the mobile agents. This service is also responsible to create the corresponding mobile agent. In this way, Mobile Grid Services are realized as this type of grid services which distribute the actual working tasks to mobile agents. Service mobility is achieved by those mobile agents.

The advantage of this design is that the existence of the relay part makes the

Mobile Grid Services conform to the Globus grid service architecture. At the same time, the mobile agent part is able to exploit useful resources in other grid nodes. Comparing with the snapshot approach [15], this approach does not waste computation on producing snapshots periodically and it takes advantage of the agent's autonomy feature. By using mobile agents with weak mobility, our proposed framework does not require any modified Java Virtual Machine (JVM).

4.1.2 Service Sharing and Discovery

In this framework, the Service Sharing and Discovery mechanism is based on the Globus Monitoring and Discovery services. It follows the approach using Web Service Protocols [24]. Mobile Grid Services are described in the Web Services Description Language (WSDL) [4] and then can be published/searched via a dynamic registry called Index Service [26] (similar to the UDDI [23]).

Since the mobile agent part of the service may migrate to any grid nodes, the relay part should have the means to know the location of its corresponding agent for communication. The Agent Management System residing on a JADE main container is responsible for this and maintains a directory of agent identifiers.

This approach avoids the centralized registry problem and supports interoperability over a distributed heterogeneous Grid environment.

4.1.3 Security

The authentication and authorization of this framework utilizes X.509 certificates [12]. It is used to guarantee the identities of users, services or resources and provide certain identities some permissions of accessing certain services or resources. This is realized by merging the corresponding JADE and Globus security mechanisms. Besides authentication and authorization, the messages exchanging between service providers and clients as well as those between mobile agents can be protected by signature and encryption. In this sense, the message integrity and confidentiality are preserved.

Similar to the approach using delegation documents [25], the resources are protected by ensuring the users and applications are trusted. Since delegation documents are not used, resource providers are less flexible on deciding the application execution permission in this framework. However, it is compensated by the reduction of extra computation and developer's effort on the delegation documents.

4.2 Overall architecture

Fig. 4-1 shows the overall architecture of this framework. A grid is composed of a group of machines (grid nodes) sharing their resources. This can be achieved by using Globus Toolkit 4 (GT4) [8]. In each grid node, a globus container should be set up to provide a hosting environment for any grid services.

The mobile grid services middleware is an add-on to the Globus grid architecture. To support the service migration which is missing in GT4, our middleware should be installed on top of the Globus container.

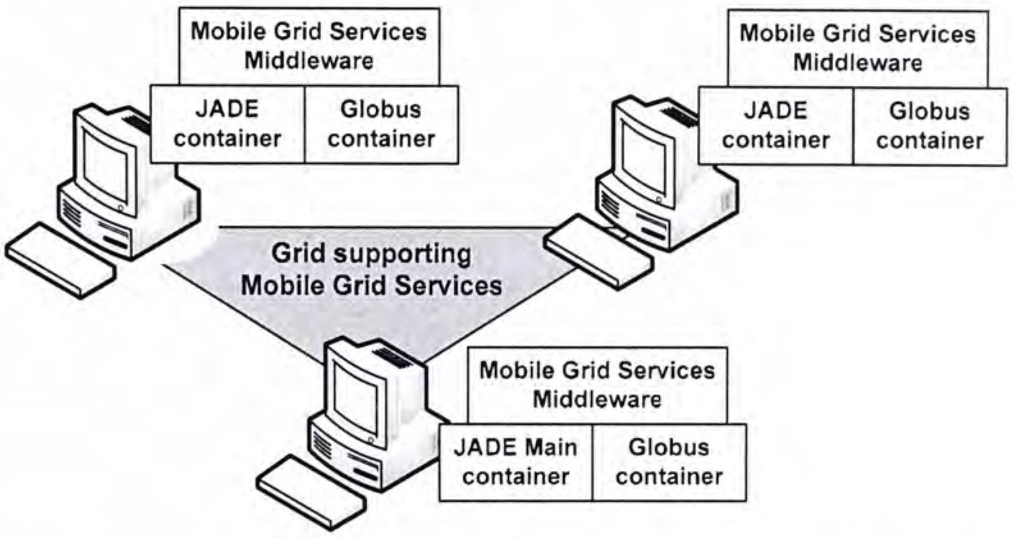

Mobile Grid Services Middleware
JADE container
Globus container
Mobile Grid Services Middleware
JADE container
Globus container
Grid supporting Mobile Grid Services
Mobile Grid Services Middleware
JADE Main container
Globus container

Figure 4-1: Overall architecture of Mobile Grid Service Framework

Since our mobility solution deploys mobile agent technology, the JADE mobile agent system [14] is used to provide agent management and migration facilities to the grid services. For each grid node supporting Mobile Grid Service (providing resources for services to migrate on them), a JADE container should be launched on it. They act as the run-time environment for the execution of JADE agents.

One of these JADE containers is called the main container which has extra agent coordination work. All JADE containers must connect to the JADE main container to form a single JADE platform. Service migration can then be accomplished by JADE agents in the Mobile Grid Services as they move throughout the grid by their intra-platform mobility.

4.3 Components of Mobile Grid Services

For any Grid Service, the service interface, the service deployment descriptor and the service implementation are essential elements that define all the details of the service. In Globus Toolkit 4 (GT4), the service interface and the service deployment descriptor are described in the Web Services Description Language (WSDL) [4] and the Web Service Deployment Descriptor (WSDD) respectively. Service implementation is a Java class or a group of classes expressing the service task logic.

Since Mobile Grid Services are realized as Globus grid services with JADE mobile agent support, WSDL and WSDD definitions (describing the service interface and the service deployment descriptor) of Mobile Grid Services are just similar to those normal Globus grid services (without mobility). However, the situation is different in the service implementation. Instead of implementing the service by Java classes directly, Mobile Grid Services implementation involves both Java classes and JADE mobile agents. The agents are implemented by extending the *Agent* class JADE libraries such that they inherit the abilities of migration, message transmission, etc. The three main components in the implementation part of Mobile Grid Services are Agent Manager, Task Agent and Monitor Agent.

4.3.1 Agent Manager

For each Mobile Grid Service, there is one and only one Agent Manager. It is responsible for managing the agents used for its own service. Besides, it acts as a relay between the client and the agents in the service. Agent Manager is not a JADE mobile agent and thus it lacks of mobility. It is stationary and resides on the initial node. This feature avoids the redeployment of the service and keeps the original Globus architecture.

The main duties of Agent Manager are to:

- Create Task Agent and Monitor Agent (if necessary)
- Receive client requests through Grid Service calling
- Redirect client requests to the Task Agent through ACL messages
- Redirect results from the Task Agent to client
- Store execution results of tasks for inquiry from clients
- Keep track of the Task Agent and Monitor Agent locations as well as their execution situation
- Send special command messages to the Task Agent and Monitor Agent such as forcing agent migration and termination.

4.3.2 Task Agent

It is a JADE mobile agent which is responsible for the actual service task of the Mobile Grid Service. It possesses the ability of migration such that it can move to other hosts for execution. If automatic migration is needed, each Task Agent will have one Monitor Agent working with it. Task Agent is created by Agent Manager in the service and each service may have multiple Task Agents for different tasks.

The main duties of Task Agent are to:

- Carry out the actual tasks of the application services
- Perform suitable actions after receiving ACL command messages from Agent Manager and Monitor Agent
- Send intermediate and final results of the task to Agent Manager
- Send ACL message to stop the execution of the corresponding Monitor Agent (if any) after the task finishes

4.3.3 Monitor Agent

It is a JADE mobile agent which is responsible for monitoring the resource information of the grid nodes and make migration decisions. It is used to help the corresponding Task Agent to do monitoring and migration decision work. This design allows the Task Agent to concentrate on its actual service task while the Monitor Agent handles resource information. No Monitor Agent will be created if the service providers do not want their Task Agents to carry out any automatic runtime migration. Service provider can adjust the migration decision policy in the Monitor Agent to fulfill his requirement.

The main duties of a Monitor Agent are to:

- Receive grid nodes resource information from the Resource Information Service
- Store the resource information of different nodes
- Use its own logic to analyze the resource information
- Make the migration decision including "move or not" and "where to move"
- Send an ACL message to the corresponding Task Agent and order it to move to suitable situations
- Go and work along with the corresponding Task Agent (i.e. Monitor Agent always follows the migration of the corresponding Task Agent)

4.4 Resource Information Service

One of the main advantages of Mobile Grid Services is the automatic run-time service migration when the home host is running out of resources. Therefore, resource information (configuration and current usage) of grid nodes is essential in our framework for supporting the automatic migration. In our framework, the information is provided by the Resource Information Service.

The raw resource data of the node machines including processor and memory information, CPU and memory usage and host data are collected from the Ganglia cluster monitoring system [27] via the Globus Index Service [26] regularly. Besides resource information, this service also gets all the Monitor Agents' details from the Directory Facilitator (DF) residing on the JADE main container. The agent information is used to find out suitable audiences (i.e. all Monitor Agents in the grid) for the resource information.

The collected resource information will not be redirected to Monitor Agents directly. The irrelevant information will be filtered out first. In order to reduce the message size used for resource data transmission, the resource information will be further processed to calculate some metric values for each node before delivery. After the calculation, the processed results will be sent to all Monitor Agents through ACL messages. The Monitor Agents for different Mobile Grid Services can then use the received resource information to make their migration decisions.

4.5 Scenario of Mobile Grid Service Execution

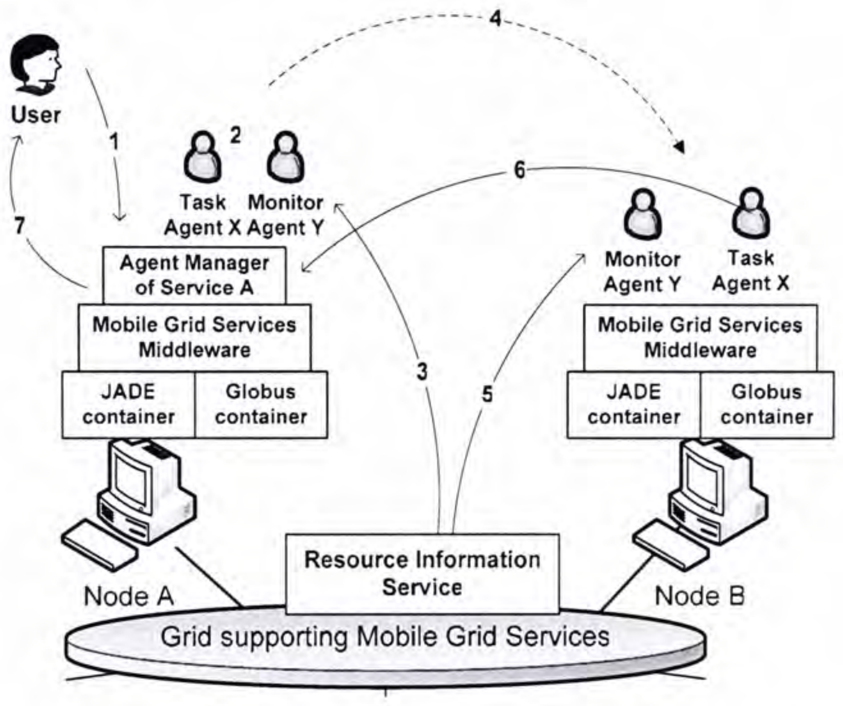

User
1
2
3
4
5
6
7
Task Agent X
Monitor Agent Y
Agent Manager of Service A
Mobile Grid Services Middleware
JADE container
Globus container
Monitor Agent Y
Task Agent X
Mobile Grid Services Middleware
JADE container
Globus container
Node A
Resource Information Service
Node B
Grid supporting Mobile Grid Services

Figure 4-2: Execution of a Mobile Grid Service

Fig. 4-2 shows the steps involved when a Mobile Grid Service executes in the framework. Assume that a Mobile Grid Service "Service A" is setup on Node A. Its Agent Manager listens to any user's requests. The steps are as follows:

1. User requests an operation in "Service A" (e.g. a complex computation).
2. Agent Manager of "Service A" receives the request and creates appropriate Task Agent and Monitor Agent (called X and Y respectively) on Node A.
3. Task Agent X starts its service task while Monitor Agent Y starts to receive any resource information from the Resource Information Service.
4. Resources in Node A are running out due to some reasons. Monitor Agent Y gets the information and decides to migrate to Node B which has plenty of idle resources. Then, it moves to Node B with Task Agent X.

5. Both agents arrive at Node B. Task Agent X resumes its work while Monitor Agent Y continues to receive information from the Resource Information Service.
6. Task Agent X completes the service task. Agent Manager gets the result of the operation and the two agents terminate.
7. The result of the operation is redirected to User from Agent Manager. In fact, User can ask for the execution situation of the service from the Agent Manager throughout the task execution.

Chapter 5

MGS API

This chapter presents an application programming interface called MGS which supports service development in the Mobile Grid Service middleware framework. The API consisting of the AgentManager Class, Task Agent templates and configurable Monitor Agent is used to provide both an easy and flexible environment for Mobile Grid Services development. The security support of the framework in the MGS API will be presented in Chapter 6 but not in this chapter.

5.1 API design

In this section, the main issues considered in the design of the MGS API will be presented. In the Mobile Grid Service Framework illustrated in Chapter 4, each Mobile Grid Service is composed of the Agent Manager, Task Agent and Monitor Agent components. Obviously, the service implementation is more complex than those of standard Grid Services. The problem becomes more significant as mobile agent technology is employed. The deployment of Mobile Grid Services will be discouraged if developers need to fully implement all these components.

In order to relieve this drawback, the MGS API is built with the aim of supporting easy Mobile Grid Services development. It is achieved by providing a collection of methods for Mobile Grid Services development. Complex operations for agent management and communication are hidden from service developers by wrapping them into some simple methods.

On the other hand, the API should be flexible enough for developers to implement their services without too many constraints. Though different services may have different requirements on their design, developers should be able to use the API to implement them. Therefore, maintaining high flexibility is another important issue for the API design. Many parts of the API are specifically designed for overcoming this issue. For example, the decision logic of the Monitor Agent can be configured with different parameters in order to utilize suitable migration strategies. Developers can even make use of self-defined logic by modifying the Monitor Agent in simple steps.

5.2 API Implementation

5.2.1 Overview

Fig. 5-1 shows the architecture of the MGS API. The API is written in the Java programming language [28]. It is built on libraries of the Java Agent Development Framework (JADE) [14] for the implementation of mobile agent management and communication. At the same time, it will use some of the facilities in the Globus Toolkit 4 [8].

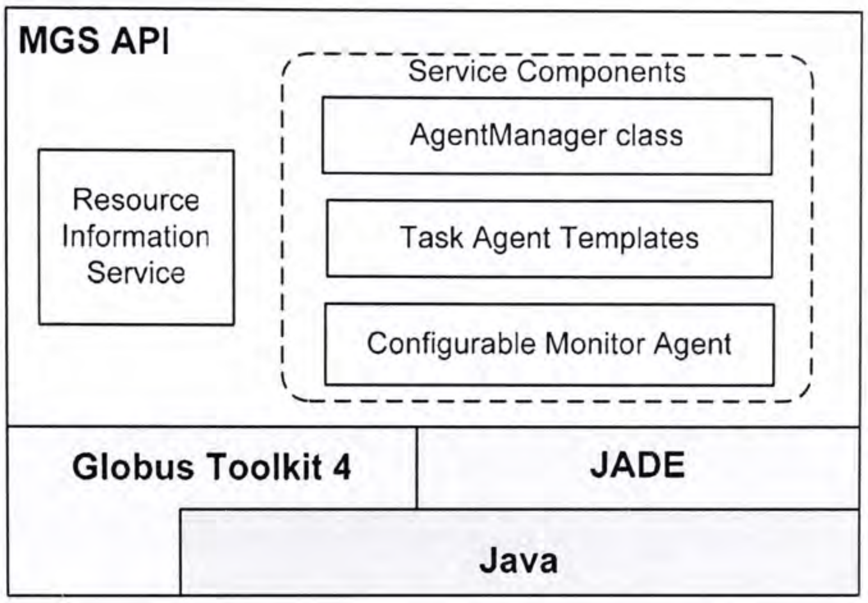


Figure 5-1: Architecture of the MGS API

This API provides a collection of libraries for supporting the development of Mobile Grid Services. As mentioned in the previous chapter, Mobile Grid Services are realized as special grid services which distribute the actual working tasks to mobile agents. The main components of each Mobile Grid Service including Agent Manager, Monitor Agent and Task Agent can be easily implemented with the help of the MGS API.

Generally speaking, the API can be divided into four parts: AgentManager Class, configurable Monitor Agent, Task Agent Templates and Resource Information Service. AgentManager Class takes the role of the Agent Manager component and provides methods for developers to manage the created agents. Configurable Monitor Agent is the default implementation of the Monitor Agent component and provides configurable options to meet various requirements. For the Task Agent component, developers must implement their own versions according to their requirements. Task Agent Templates are provided in the API to help developers to do this job in a simpler way. Resource Information Service is implemented for providing resource information in order to support automatic runtime service migration. Detailed descriptions of the four components are given in the followings.

5.2.2 Agent Manager Class

In a Mobile Grid Service, the Agent Manager component (non-agent part) basically acts as a relay between users and the mobile agents (i.e. Task Agents and Monitor Agents). This part of the Grid Service implementation can be simplified and partially accomplished by employing the *AgentManager* class in the MGS API.

In the Grid Service implementation, an *AgentManager* object should be created to enable the service to possess the essential functions of Agent Manager. These functions include managing agents used in the service, preparing the execution environment connected to the JADE main container and setting up communication channels for those agents.

During the startup of the *AgentManager* object, a JADE remote container (connecting to the JADE main container) is setup by using the JADE in-process interface. This interface allows an external Java application to use JADE as a kind of library. That means it allows the launching of the JADE Runtime as well as the creation of agents within the application itself. The JADE container in the *AgentManager* object provides an executing environment for all newly created agents through the Agent Manager. Besides the container, an Agent Manager Agent is also created during the setup of *AgentManager* object. This agent is used to help the *AgentManager* object to send ACL messages to and receive messages from other agents in the JADE platform. It is stationary because it always resides at the same machine as the static Agent Manager and thus it is not necessary to move.

Internal communication mechanism To preserve the autonomy of agents, the in-process interface of JADE is designed such that the application cannot obtain a direct reference to the agents and cannot perform method calls on the agents. That means the *AgentManager* object is unable to pass its commands or queries to the Task Agents through direct method calls on the Agent Manager Agent. Therefore, a special mechanism is taken to carry out the message exchange between Agent Manager and Agent Manager Agent.

Fig. 5-2 shows the communication between *AgentManager* object and Agent Manager Agent. For *AgentManager* object sending a message to Agent Manager Agent, the object-to-agent channel provided by JADE is used. This channel allows applications to pass objects to the agents they hold. In our implementation, Agent Manager Agent will enable this object-to-agent communication channel and continuously receive any objects from the *AgentManager* object. Once *AgentManager* object needs to send an ACL message, it will pass the message as well as the reference of an *ArrayBlockingQueue* object to its Agent Manager Agent. The agent will then send the message to the target agent and wait for a reply. After receiving the reply, the message is needed to flow in the opposite direction. The Agent Manager Agent will put the reply message to the received *ArrayBlockingQueue* object. The *AgentManager* object will be notified that the queue is non-empty and get the reply ACL message. By this mechanism, *Agent Manager* can carry out the message exchange with Agent Manager Agent successfully and further communicate with all agents in the JADE platform.

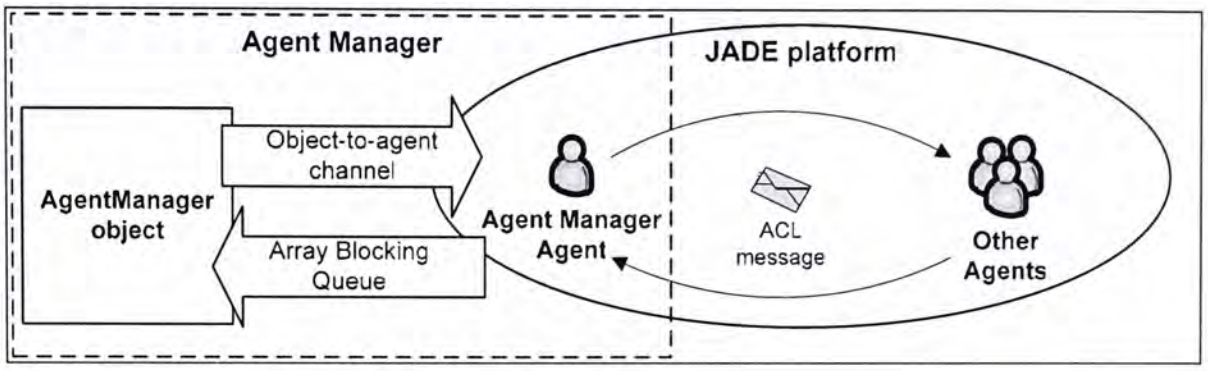


Figure 5-2: Structure of Agent Manager

ResultTable mechanism The ResultTable mechanism is used for the Task Agent to send back the execution result to Agent Manager. Normally, Task Agent starts to execute after receiving client request through the Agent Manager. The execution result can be sent in the reply to the client request such that the client can get the result from the returning of *sendACL()* method. However, since the execution time of a Task Agent may be very long, it is unreasonable if returning from the *sendACL()* method is the only way to get the result. As a result, Task Agent should have another means to send the result back to Agent Manager anytime.

To achieve it, a ResultTable is maintained in the *AgentManager* object. This table is a hash table which stores a number of *MGSResult* objects with task name as their keys. Each *MGSResult* is composed of a task result (in form of *String* object) and a flag "*isFinal*" indicating if the result is finalized.

After executing to a certain stage, Task Agent can send the temporary result to its corresponding Agent Manager Agent by invoking *notifyResult()* method. It will send the result through a notification (DF notification in JADE) to the Agent Manager Agent. After receiving the notification, an *MGSResult* object (with the flag "*isFinal*" as false) will be created and replace the previous one in the ResultTable.

When the clients want to get the current result of their tasks and invoke *getCurrentResult()* method in the Agent Manager, the corresponding *MGSResult* object obtained from the table will be returned.

When the Task Agent finishes the execution, it can send the final result to the Agent Manager by invoking the *notifyFinalReuslt()* method. A final notification (used for notifying the end of the task execution) will be received by the Agent Manager Agent. It will then replace the corresponding record in the Result Table by a new *MGSResult* object (with the flag "*isFinal*" set as true).

Moreover, the *MGSResult* object contains the "*startTime*" and the "*endTime*" variables. The "*startTime*" is recorded at the moment of creating the Task Agent. The "*endTime*" is recorded when the final notification is received from the Task Agent. Furthermore, there is an "*isChecked*" flag in the *MGSResult* object which will be used for the agent protection support (details in Chapter 8).

The ResultTable mechanism provides a means for the Task Agent to notify the temporary results in different execution stages to the client. Besides, the client does not need to keep the connection with Agent Manager through the whole execution time. He can be offline after sending the request and then ask Agent Manager for the execution result anytime (even after the termination of the Task Agent) afterwards.

Provided methods The *AgentManager* class also provides methods to help service developers to implement agent management and interaction between users and services in their Mobile Grid Services. The main methods provided are:

- void createAgent(String agentName, String agentClass, Object[] agentArgs)

  This method is used to create a new Task Agent in the Mobile Grid Service. Since different tasks should be implemented in different Task Agents, this method receives an argument *agentClass* in order to allow developers to specify their required Task Agent being instantiated in their services. No Monitor Agent will be created by this method. It is designed for those services which do not require automatic service migration initiated by the Monitor Agent. The mobility of the Task Agent is preserved such that its execution still can move to other hosts if the route is preset.

- void createAgentWithMonitor(String agentName, String agentClass, Object[] agentArgs, String monitorClass, Object[] monitorArgs)

  Unlike the *createAgent* method above, this method will create the specified Task Agent as well as a Monitor Agent in the service. The argument *monitorClass* allows the developer to specify the Class used to instantiate the Monitor Agent. Another new argument *MonitorArgs* is used to pass arguments to the Monitor Agent.

- void createAgentWithMonitor(String agentName, String agentClass, Object[] agentArgs, MonitorSetting setting)

  This method is similar to the one above but the Class used to instantiate the Monitor Agent is fixed. It will use the default Monitor Agent provided in the MGS API compulsorily such that developers have no need to be concerned about the Monitor Agent class. The argument *MonitorSetting* is used to configure the default Monitor Agent. Details of the default Monitor Agent are shown in section 5.2.4.

- ACLMessage createACLMessage(String agentName, int performative, String content)

  It is used to create an ACL message for the message exchange between the Agent Manager and mobile agents. The arguments are used to specify the receiver, the content and the FIPA performative of the message (e.g. REQUEST, INFORM).

- String sendACL(ACLMessage query)

  It is used to send a command or a query to mobile agents through an ACL message from the Agent Manger. This method is important for implementing the interactions between service callers and Task Agents. After being created by the *createACLMessage* method, the ACL message can be sent to the target agent through this method and the reply will be returned. In the meantime, the ACL message is passed from the *AgentManager* object to the Agent Manager Agent while the reply message is forwarded in opposite direction (as shown in Figure 5-2). The actual sending and receiving procedures are hidden in this method. Service developers can concentrate on the design of ACL message exchange and the handling of the reply (i.e. return of *sendACL()* method).
- MGSResult getFinalResult(String taskName)

  This method is used to get the final result of a Task Agent from the ResultTable. It will call the getCurrentResult() method repeatedly until the obtained result is finalized. The status of the result can be identified by checking the "isFinal" flag of the returned MGSResult object. Finally, the final result of the task will be returned.

- AgentSituation getAgentSituation(String agentName)

  This method returns the current situation of a specific agent. The result is returned as an *AgentSituation* object containing information about agent state, position, etc. In fact, a special ACL message is sent to the agent to ask for the current situation.

- void moveAgentTo(String agentName, String containerName, String containerAddress)

  This method directly forces the agent named with the specified *agentName* to move to a specific host. The command is sent to the agent platform instead of the agent itself.

5.2.3 Task Agent Templates

The Task Agent component is the most variable part in the Mobile Grid Services. As it is responsible for the actual service task, the implementation of the Task Agent is heavily dependent on the task properties of the Mobile Grid Services. It is impossible to have a single universal Task Agent which can fulfill all requirements of various types of service tasks. Therefore, the implementation of Task Agent must rely on the developers themselves.

In the MGS API, the basis of the Task Agent component is the TaskAgent class. The *TaskAgent* class is the main body of the Task Agent. It extends the *Agent* class in the JADE library to inherit the ability to accomplish essential interactions with the agent platform (e.g. registration) and communication with other agents (through ACL message exchange). It is also responsible for the initialization of the agent. The *TaskAgent* class provides a series of methods for the basic actions of Task Agent. It provides sendMessage() and receiveMGS() for sending and receiving ACL messages in the MGS framework respectively; moveMGS() for moving the agent to a specified host. On the other hand, two methods - *notifyResult()* and *notifyFinalResult()* are provided to notify (via DF service notification in JADE) the temporary or final result to the Agent Manager Agent (usually after Task Agent finishes its task to certain stage).

However, the most important tasks (the message handling and the actual service task) are not implemented in the TaskAgent class. As stated in the implementation procedure of JADE agents, the developer who wants to implement an agent-specific task should define one or more *Behaviour* (in JADE libraries) subclasses, instantiate them and add the *Behaviour* objects to the agent job list.

To help developers building application-specific Task Agents for their own Mobile Grid Services, two templates are provided. A template consists of several incomplete classes which have already implemented most of the basic and essential elements in a Task Agent. To complete the Task Agent implementation, developers need to follow the guidelines in the template and complete all the classes such that they can carry out their application-specific tasks. The details of the two Task Agent Templates provided in the MGS API will be presented below.

TaskAgent Template This template is composed of three Java files. The details of the template and the Java files are as follow:

- *MyTaskAgent.java*

  It extends the *TaskAgent* class and it will become the main body of the

Task Agent. It is responsible for the declaration and implementation of any instance variables and methods in the Task Agent. Besides, the *TaskServeIncomingMessageBehaviour* object is created and added to this class.

- *TaskServeIncomingMessagesBehaviour.java*

  This Behaviour class is responsible to receive any incoming ACL messages and keeps reacting with all relevant messages. For instance, it will start the migration process and move to the target host marked in the message when it receives a "move" message from the Monitor Agent. Hence, all the message handlings should be defined in this class. To simplify the implementation, all the handling of pre-defined command messages such as "move" is already implemented in the template.

- *TaskBehaviour.java*

  It is the class dealing with the actual service task. Developers should implement the program logic of the service tasks in the *action()* method in this class. The *action()* method will be invoked when the Task Agent starts. After *action()* returns, the *done()* method will be called. The return value of this *done()* method will determine the life of the *Behaviour* class. If the value is true, the execution will end. Otherwise, the *action()* method will be invoked again. Therefore, the stopping criteria of the execution should be implemented in the *done()* method.

To develop a new Task Agent, a developer should implement each service task into the indicated location of the *TaskBehaviour* class. Different tasks should be implemented in different *TaskBehaviour* classes. Then, these *Behaviour* classes should be added to the *MyTaskAgent.java* by following the guideline in the template. Although, the handling of pre-defined command messages are already implemented in the template, service developers still are responsible to implement the reaction of receiving their self-defined command messages from the Agent Manager or other message exchange with other agents. Finally, the user-defined instance variables and methods should be coded in the *MyTaskAgent.java* (following the guidelines).

SimpleTaskAgent Template This template contains one Java file only. Before describing the *MySimpleTaskAgent.java* in the template, the *SimpleTaskAgent* class (in the MGS API) utilized by the template will be presented first.

The *SimpleTaskAgent* class is implemented for further simplifying the work of developers during their own Task Agent implementation. It extends the *TaskAgent* class to inherit all the essential abilities and methods for the Task Agent component. In addition, it contains three abstract methods:

- public abstract void normalExecution()

  This method states the program logic of the service tasks.

- protected abstract void msgHandler()

  This method will be invoked for each iteration. The handling of any incoming ACL messages should be implemented. Template is provided where handlings of all essential messages for MGS are implemented. Developers only need to manage the user-defined messages.

- protected abstract boolean endChecking()

  This method is used to specify the stopping criteria of the Task Agent execution. If the stopping criteria are met, it should return true.

Unlike the *TaskAgent* class without any *Behaviour* object, a *Behaviour* object is created and added to the *SimpleTaskAgent* class such that its *action()* method with be invoked repeatedly. This causes the *msgHandler()* method and the *normalExecution()* method to be invoked once in each iteration. At the end of each iteration, the *endChecking()* method (defined in the *SimpleTaskAgent* class) will be invoked to check whether the task execution should be finished. The *action()* methods will be invoked repeatedly until the *endChecking()* returns true (i.e. stopping criteria is met). Before the execution ends, it will send an ACL message to stop the corresponding Monitor Agent (if any).

To use the SimpleTaskAgent Template to implement a Task Agent, only the *MySimpleTaskAgent.java* file is required to be considered. It extends the *SimpleTaskAgent* class and thus the three abstract methods should be implemented. The developer needs to implement the service task in the *normalExecution()* method, specify stopping criteria in the *endChecking()* method and implement the user-defined message handling in the *msgHandler()* method following the guidelines. The handlings of all essential messages for MGS are already implemented in the template. Besides, the user-defined instance variables and methods should be written in the indicated location. The developer has no need to handle any *Behaviour* object because it is already implemented in the *SimleTaskAgent* class. Moreover, it is more convenient to implement a Task Agent by managing a single file only. If a Task Agent is implemented by using this template, it can be converted to a "protected" version which supports agent protection (details in Chapter 7). However, the drawback of this template is that the structure of the created Task Agent is less flexible (e.g. only one *Behaviour* object can be added to the agent).

5.2.4 Configurable Monitor Agent

A default Monitor Agent is provided in the API. It is used to accomplish the implementation of the Monitor Agent component in a Mobile Grid Service. The presence of this configurable Monitor Agent allows service developers to ignore the related implementation and thus simplifies the development process. The basis of the Configurable Monitor Agent is *MonitorAgent* class which is implemented by extending the *SimpleTaskAgent* class. The implementation details of the *MonitorAgent* class including the three abstract methods (*msgHandler()*, *endChecking()* and *normalExecution()*) are discussed as follows:

Constructor To make the default Monitor Agent useful under different situations (different developers' requirements), it is designed to be configurable. This goal is achieved by the *MonitorSetting* object. The constructor of *MonitorAgent* class will take in a *MonitorSetting* object as argument. The *MonitorSetting* object contains some configurable variables for controlling the execution mode (normal mode and debug mode) of the Monitor Agent. In debug mode, the Monitor Agent will show detailed information about any incoming and outgoing messages as well as migration decisions on the screen for debugging purposes. In normal mode, the Monitor Agent will keep silent (unless error occurs) to reduce unnecessary overhead and hide the existence of the Monitor Agent from the service/resource provider. Besides the execution mode, the migration decision logic in the Monitor Agent (in the *decide()* method) can be configured by the variables in the *MonitorSetting* object. The service developers can set the minimum requirements and the importance weights of each kind of resources (CPU, RAM, and HD). Finally, the Monitor Agent will register to the JADE platform with service type as "monitor-agent". This allows the Resource Information Service to recognize it during the resource information distribution.

Instance variables The Monitor Agent will store the resource information for migration decisions. A set of hash tables are created in the *MonitorAgent* class to store those resource data (including the CPU value, RAM value and HD value calculated by the Resource Information Service) for each host.

Private method The *MonitorAgent* class contains a *decide()* method which is responsible for analyzing the resource information and make migration decisions. It will decide whether migration is necessary and where is the best place to move to. The *decide()* method does not take in any argument. It uses the information stored in the hash tables for migration decisions. It will return the chosen host name (if decide to move) or null (if decide not to move).

In the *decide()* method, the migration decision is made after a series of procedure is completed. The first step is to check whether the current host meets the minimum resource requirements. The requirements represent the least CPU, RAM, and HD values for the current host (amount of the currently available CPU, memory and harddisk in the current host) such that the agent has no need to consider the migration. They can be set by the service developers through the variables in the *MonitorSetting* object. If the minimum requirements are met, no migration is required and the *decide()* method will return null.

The second step of the migration decision is to decide the best host in the grid at that time according to the current resource information. To compare the available hosts, a "Score" value will be calculated for each host by the following formula:

Score = (CPU_weight * CPU) + (RAM_weight * RAM) + (HD_weight * HD)

The values "CPU_weight", "RAM_weight" and "HD_weight" can be set in the *MonitorSetting* object such that the importance weight on CPU, memory and harddisk used in the decision logic can be configured by the service developers. For example, the CPU values will be more important in the calculation of "Score" if the importance weight of CPU is higher than the others. After comparing the "Score" values of all hosts, the host with the highest value will be marked as the best host.

The third step is to further confirm the necessity of the migration. Although the best host is chosen in the previous step, it does not mean that an agent would get benefits after moving to the best host. If the agent moves to a host with very similar resource usages to those in the current host, it cannot get a significant improvement on the execution environment. On the other hand, the execution time may be prolonged due to the overheads induced by the worthless migration. Therefore, three "Gain" values will be calculated to find out the expected improvement on each kind of resources. It is done by comparing the resource values (received from the Resource Information Service) of the current host and the best host. The formulas are as follows:

Gain_CPU = (best_CPU – current_CPU) / current_CPU

Gain_RAM = (best_RAM – current_RAM) / current_RAM

Gain_HD = (best_HD – current_HD) / current_HD

Migration will be carried out only when any "Gain" value (Gain_CPU, Gain_RAM or Gain_HD) is greater than five percent. This makes sure that the migration will bring a certain gain to the agent. After this procedure, the migration decision is completed and the *decide()* method will return the decision result.

Although the provided configurable options make the decision policy of the migration more flexible, they still cannot fulfill all developers' requirements undoubtedly. For example, developers may require that certain hosts have extra bonus score in the migration decision. For any special requirements (like the above example) on the migration decision logic, service developers can also develop their own Monitor Agent by extending the provided one. The only work required is to override the *decide()* method. In this way, they can produce their own Monitor Agents which are best-fit to their requirements

The msgHandler() method The *msgHandler()* in the *MonitorAgent* class is responsible to handle the ACL messages received from the Resource Information Service. When message containing resource information is received, the received data includes the host name, the CPU, RAM and HD values will be used to update the hash tables in *MonitorAgent* class. When "decide" message is received, the *decide()* method will be invoked. If the *decide()* method returns a host name, it will send "move" message to the corresponding Task Agent and then move to the decided host together.

The endChecking() method The end of Monitor Agent's execution is not decided by itself but depends on when the corresponding Task Agent finishes its task. Monitor Agent will stop the execution after receiving the "end" message from its corresponding Task Agent. Therefore, the *endChecking()* method in the *MonitorAgent* class is implemented to return false all the time.

The normalExecution() method The main work of the Monitor Agent is receiving resource data from the Resource Information Service and then making migration decisions. They are completely implemented in the *msgHandler()* method. Therefore, the *nomralExecution()* method for *MonitorAgent* class is empty.

5.2.5 Resource Information Service

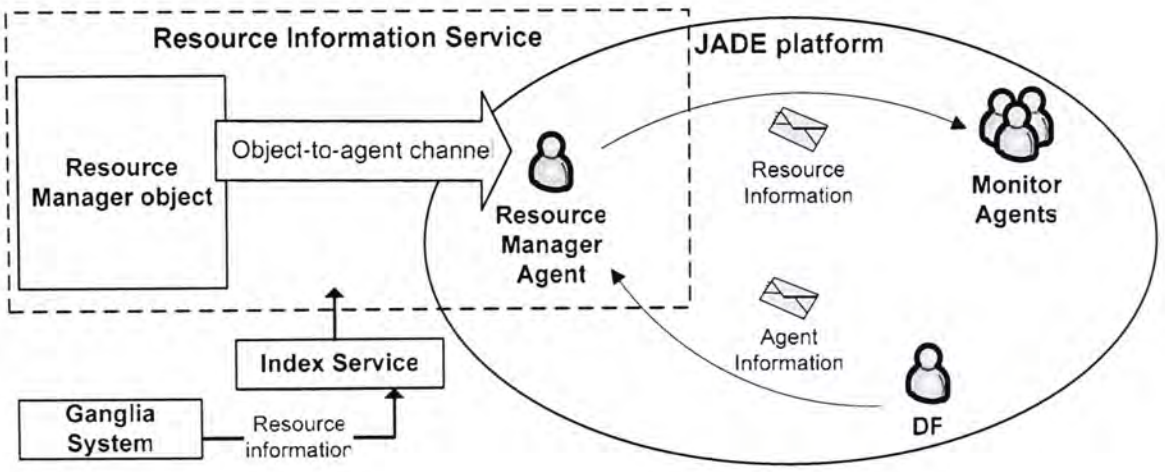


Figure 5-3: Structure of Resource Information Service

Resource Information Service is responsible for providing resource information in our framework. It is implemented in the form of standard Grid Services. In this section, the implementation details of the Resource Information Service will be presented.

From Fig. 5-3, we can see that the service contains a Resource Manager Agent and a Resource Manager object. The Resource Manager Agent is a static agent which will never perform any migration. It is used to obtain agent information from the JADE platform and send ACL messages to Monitor Agents. On the other hand, the Resource Manager object is used to pass command to and communicate with the Resource Manager Agent.

The source of the resource information used in the Resource Information Service is the Ganglia cluster monitoring system [27]. After installing the Ganglia system on a host, the raw resource data of the machine including processor and memory information, CPU and memory usage and host data can be collected. Therefore, all machines in the grid must be installed with the Ganglia system in order to generate resource data for all grid nodes. The resource data will then be obtained by a Globus Index Service [26] called Default Index Service. The Default Index Service will be set up automatically when the Globus container is created. It obtains the generated data from the Ganglia system regularly and makes the resource information accessible from others. Originally, the resource information in the Default Index Service is updated only every five minutes which is too infrequent for our framework. After modification, it will update the resource information every 60 seconds.

The Resource Information Service gets the resource information by querying the Default Index Service about the value of resource property "Entry". It is repeated every 60 seconds such that the information can be updated repetitively. Fig. 5-4 shows the resource data of a host (in form of XML file [3]) received from the query.

The collected information will then be processed under a series of procedures.

The first step is filtering out the irrelevant information. In our current implementation, we only focus on the information of CPU clock speed, last minute CPU usage, current available RAM size and available harddisk space. According to the configuration of the Resource Information Service, only resource data of the specified hosts will be considered. If no host is specified in the configuration, the information of all available hosts will be considered. The resource information will be further processed to calculate three values (CPU, RAM, HD) before transferring to the Monitor Agents. The treatments on the data are done in the Resource Information Service in order to reduce the message size used for resource data transmission and avoid duplicate calculations in every Monitor Agents.

```
<ns1:Host ns1:Name="dell04" ns1:UniqueID="dell04">
      <ns1:Processor ns1:CacheL1="0" ns1:CacheL1D="0"
            ns1:CacheL1I="0" ns1:CacheL2="0"
            ns1:ClockSpeed="3192"
            ns1:InstructionSet="x86"/>
      <ns1:MainMemory ns1:RAMAvailable="17"
            ns1:RAMSize="2024" ns1:VirtualAvailable="2114"
            ns1:VirtualSize="4169"/>
      <ns1:OperatingSystem ns1:Name="Linux"
            ns1:Release="2.6.9-1.667smp"/>
      <ns1:Architecture ns1:SMPSize="4"/>
      <ns1:FileSystem ns1:AvailableSpace="53712"
            ns1:Name="entire-system" ns1:ReadOnly="false"
            ns1:Root="/" ns1:Size="63544"/>
      <ns1:NetworkAdapter ns1:IPAddress="192.168.0.14"
            ns1:InboundIP="true" ns1:MTU="0"
            ns1:Name="dell04" ns1:OutboundIP="true"/>
      <ns1:ProcessorLoad ns1:Last15Min="20"
            ns1:Last1Min="149" ns1:Last5Min="57"/>
</ns1:Host>
```

Figure 5-4: Resource data (XML file) received by Resource Information Service

Before presenting the calculation of the CPU, RAM and HD values, three "maximum" variables used in the Resource Information Service will be introduced first. These three variables are Max_CPU, Max_RAM and Max_HD which are the best possible/expected CPU ClockSpeed, RAM available and Harddisk space available in the whole grid respectively. These variables are configurable during the setup of the Resource Information Service and the grid managers can adjust suitable values for their own grid according to the hardware available and the expected service requirement. Table 5-1 shows the unit used in the three "maximum" variables and an example configuration in a grid:

Table 5-1: EXAMPLE VALUE OF THE THREE "MAXIMUM" VARIABLES IN RESOURCE INFORMATION SERVICE

Variable Name	Example Value	Unit
Max_CPU	2800 (2.8 GHz)	Megahertz (MHz)
Max_RAM	1024 (1 GB)	Megabyte (MB)
Max_HD	1024 (1 GB)	Megabyte (MB)

Assume that Host A has a 3.0GHz CPU and the current CPU usage is 20%. Moverover, 499MB RAM is available now and there is 5000MB free space in the harddisk. In the Resource Information Service, the CPU, RAM and HD values of Host A can be calculated as follows:

CPU = (3000 * (1-20%)) / Max_CPU = 2400/ 2800 = 85.71%

RAM = 499 / Max_RAM = 499 / 1024 = 48.73%

HD = 5000 / Max_HD = 5000 / 1024 = 488% (chop to 100%)

In this case, the resource data of Host A (CPU=8571, RAM=4873, HD=10000) will be transferred to appropriate Monitors Agents.

The calculated value must be between 0 and 10000. It represents the percentage of available CPU/RAM/HD compared with the best possible/expected resources. After the calculation, the processed results will be sent through ACL messages.

For each round of resource information distribution (every 60 seconds), the service will pass some commands to the Resource Manager Agent via the object-to-agent channel in the Resource Manager. The Resource Manager Agent will perform different actions when it obtains different types of command including "prepare", "resource" and "decide".

When the "prepare" command is received, it represents the beginning of a new round of resource information distribution. The Resource Manager Agent will first determine the audience for this round of information distribution (new audience list will be prepared for each round). It is required to search for all Monitor Agents available in the JADE platform. Since all Monitor Agents will register to the JADE platform and specify their service type as "monitor-agent", the Resource Manager Agent can search for all agents with service type as "monitor-agent" by sending query to the Directory Facilitator (DF) in the JADE main container. The received Monitor Agent information will not be used to make the audience list directly. Only a portion of Monitor Agents will be selected to add into the list. The selection is done randomly according to the <SendingPercent> tag in the configuration file of the Resource Information Service. If the value stated in the <SendingPercent> tag is 30, each Monitor Agent will be selected with a 30 percent chance.

When the "resource" command is received, the processed resource data of a specific host is transferred to the Resource Manager Agent. Since each "resource" command carries information for a single host only, the agent usually receives multiple "resource" commands in every round such that resource information of all available hosts can be collected. For receiving each "resource" command, the enclosed resource data will be sent in form of ACL messages to all Monitor Agents marked in the audience list.

When the "decide" command is received, it represents the end of this round of resource information distribution. After that, a "decide" message will be sent to each Monitor Agents marked in the audience list. Monitor Agent will ignite the migration decision only when it receives the "decide" message. The Monitor Agents for different Mobile Grid Services can then use the received resource information to make their migration decisions.

5.2.6 Example Application

Fig. 5-5 shows an example of Mobile Grid Service developed on top of the MGS API. In the constructor of this Grid Service "MyApplication", an *AgentManager* object is created with the name of the configuration file (in line 5). Two methods are implemented for the client to interact with the service "MyApplication". In method1, the *createAgent()* method of *AgentManager* object is used to create a Task Agent named "task1" using the Java class in path "myPath.MyTaskAgent" (in line 8). In method2, we create and send an ACL message with content "stop_counting" to the Task Agent "task1". After that, a reply will be received as the return of *sendACL()* method (in line 11 to 13). Further handling on the reply can be done according to the service requirement.

This example demonstrates the general techniques of implementing Mobile Grid Services using MGS API. The basic steps include the creation of *AgentManager* object, using it to create Task Agents and Monitor Agents, and use other provided methods to implement any interactions between users and agents.

```
public class MyApplication{
    AgentManager manager;
    public MyApplication(){
        ...
        manager = new AgentManager("serviceSetting.xml");
    }
    public method1(){       //start Task Agent
        manager.createAgent("task1", "myPath.MyTaskAgent", null);
    }
    public method2(){       //ask Task Agent to do something
        ACLMessage msg = manager.createACLMessage("task1",
            ACLMessage.REQUEST, "stop_counting");
        String reply = manager.sendACL(msg);
        ...     // handling of the reply
    }
    ...
}
```

Figure 5-5: An example Mobile Grid Service developed using the MGS API

Chapter 6

Security Support for Mobile Grid Services

In this chapter, the security support in the Mobile Grid Service Framework is introduced. The Mobile Grid Service Framework is not practical if no security measures are provided. To complement the framework, security mechanisms consisting of authentication, authorization, message integrity and confidentiality, agent permission and agent protection are added into our framework. The details of these mechanisms except agent protection (which will be introduced in Chapter 8) as well as the security facilities provided in the MGS API will be presented.

6.1 Overview

Security is an important consideration when deploying Mobile Grid Services to real-world applications. Security measures should be included in the framework to protect the computation of the services in potentially hostile environments. Otherwise, the Mobile Grid Service Framework is incomplete and impractical. If it lacks security mechanisms, the usage and development of Mobile Grid Services will be inhibited. For example, all applications containing privacy information cannot be developed in the form of Mobile Grid Services.

To provide security support in the Mobile Grid Service Framework, we do not need to fully implement all required mechanisms. Since our framework realizes service migration by combining the Java Agent Development Framework (JADE) [14] and Globus Toolkit 4 (GT4) [8], our framework can reuse their own security mechanisms to provide security services to Mobile Grid Services. For JADE, the security support is provided by an add-on called JADE-S. It provides user authentication, agent actions authorization against agent permissions and message signature and encryption. In GT4, the part providing fundamental security services for grid services is the Grid Security Infrastructure (GSI) [11]. It is concerned with establishing the identity of users or services, protecting communications, and determining who is allowed to perform what actions.

By integrating the facilities of JADE-S and GSI into our framework, we can provide most of the essential security services for the Mobile Grid Services. Besides, we import other mechanisms to supplement the deficiencies of JADE-S and GSI and improve them. In general, four security mechanisms are provided in the framework. They are authentication, authorization, message integrity and confidentiality, agent permission and agent protection. Except for the agent protection which is presented in the next chapter (Chapter 7), the details of the other three mechanisms will be described separately in the following sections.

6.2 Authentication and Authorization

Authentication and Authorization are important in a secure grid system. They protect the Grid Services against abusing by unauthorized users. Authentication is an essential element for grid security which is the act of establishing or confirming the identity claimed by an entity. Without Authentication, the identity claimed by each entity may be faked and no identity can be trusted. Authorization is the process to check if the consumer has been granted permission to use a resource. In this section, the meaning of authorization is bound to the access right of Grid Services.

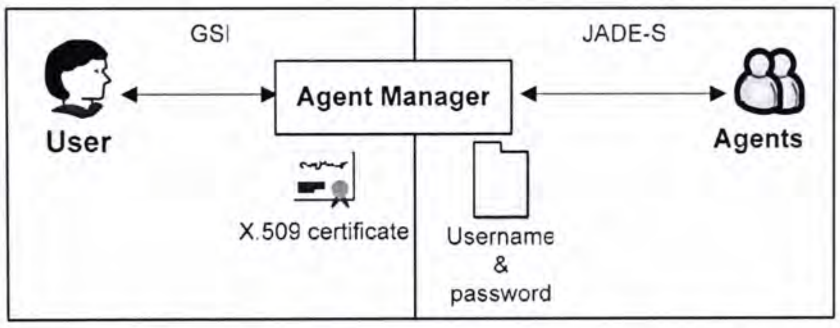


Figure 6-1: Authentication and authorization in MGS

In the framework, the authentication and authorization mechanisms for the Mobile Grid Services are mainly based on the GSI. As shown in Fig. 6-1, X.509 certificates [12] are used in GSI to guarantee the identity of the user, service and hosts during grid service requests. X.509 certificates provide each entity (user or service) with a unique identifier and a method to assert that identifier to another party through the use of an asymmetric key pair bound to the identifier by the certificate. By using the identities, the services can be configured to allow the access of clients with certain identities. Conversely, the client can choose to invoke services with certain identities only. It is important for us to make sure that users get services from valid service providers and the services are provided to valid users at the same time.

The authorization of Grid Services is not limited to use a uniform setting within the whole service. It can also be organized on a per method basis. That means each method (operation) in any Grid Services can be configured separately such that only certain users are authorized to invoke it. For example, in a grid service offering two operations (Method_A and Method_B), Method_A can only allow User_X to access whereas Method_B can allow both User_X and User_Y to invoke.

As our framework involves mobile agents, the authentication and authorization of agents in the JADE agent platform should be considered as well. The authentication and authorization mechanism in JADE-S guarantees that users creating containers and agents are known to the system. This is achieved by checking whether the user's username and password are valid or not. This prevents agents created by unknown users from abusing grid resources or illegally interacting with other working agents such as Task Agents and Monitor Agents.

Initially, each valid service provider should get their own username and password pair for the JADE platform. All these password information will be recorded in a password file in the JADE Main Container. It is used for the password checking process during authentication. Service providers need to offer their usernames and passwords when they start up the service (simultaneously login to the JADE platform). This ensures that only people possessing valid passwords (valid service providers) can login to the JADE platform. No one can login to JADE and carry out any attack unless he gets a valid username and password.

The JADE-S authentication and authorization is done by service providers only and service requestors do not need to enter passwords for their own identity for the JADE platform. This hides the presence of the JADE platform from the service users. In this design, the creation of agents in a service is on behalf of the service provider. That means that all the actions done through the service (even via agents created in the service) will be on behalf of the service provider but not the caller of the service. This is reasonable as the agent of a Mobile Grid Service is the extension of the service which should work on behalf of the service provider.

6.3 Message Integrity and Confidentiality

Message exchange is important in our framework since most of the interactions between Grid Services and clients as well as the communication among agents are based on message exchange. It will become a main security vulnerability of the Mobile Grid Services if there is no message protection.

To prevent it, we have provided mechanisms to offer both message integrity and confidentiality in the framework. Message integrity means the confidence that the message data has not been tampered with during transmission. Receiver can verify messages were not altered in transit from the sender. This can be achieved by using message signature. A mathematical hash will be computed for the message and stored in the message before sending to sender. The recipient can ensure that the received message has not been changed by computing the hash again.

On the other hand, confidentiality is the confidence that only the expected receiver will be able to read the clear message. This prevents the privacy details in the message from being disclosed by eavesdropping. This can be achieved by message encryption. The message will be first encrypted by the sender and then transmitted to the receiver containing a decryption key. The content of the encrypted message is meaningless to anyone without the correct decryption key.

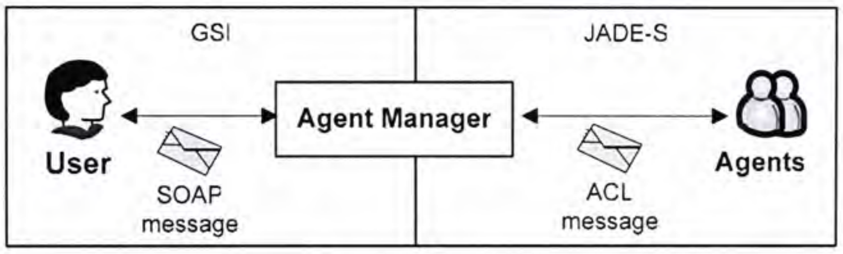


Figure 6-2: Message exchange in MGS

As shown in Fig. 6-2, the message protection mechanism is divided into two levels in our framework. It is because our framework involves two types of message exchange. One of them is the Agent Communication Language (ACL) messages for the JADE agent communication while another is the Simple Object Access Protocol (SOAP) [5] messages for the interaction between services and clients.

In JADE, agents communicate with each other by the ACL messages defined by the FIPA international standard [13] for agent interoperability. JADE-S supports signature and encryption on the ACL messages such that developers do not need to deal with the actual signature and encryption mechanisms. Developers just need to request a message to be signed (encrypted) or check whether a received message has been signed (encrypted) by calling appropriate libraries.

In GT4, SOAP is used as the Web Services message protocol for communication. All interactions between clients and services are by means of SOAP messages. To protect the integrity and confidentiality, GSI provides mechanisms for signing and encrypting the SOAP messages. This can be done by configuring the corresponding policy file (security descriptor).

Since the overhead of signature and encryption heavily depends on the message length and frequency, it may be large and affects the normal execution. Therefore, in our framework, message integrity and confidentiality are not compulsory and service providers can decide whether to use them or not according to their specific consideration.

6.4 Permissions on Agents

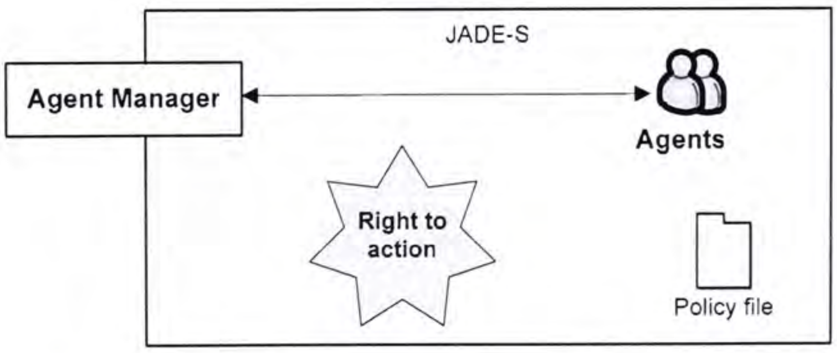


Figure 6-3: Agent permission in Mobile Grid Services

In this section, the action permission of the agents in the Mobile Grid Services will be considered. Unlike the authorization mentioned in the previous section (which is about the clients' right to access a service), this section focuses on the permission of agent actions in a service.

In our framework, permissions will be used to prevent agents in a Mobile Grid Service from abusing grid resources or improper interaction with Task Agents and Monitor Agents of other Mobile Grid Services.

Through the permission mechanism in JADE-S, we are able to assign some permissions to agents to limit their actions. This can selectively allow agents with certain identity to perform some actions or access some resources according to a set of rules described in a policy file. The format of the rules must follow the Java Authentication and Authorization Service (JAAS) syntax [29].

JAAS has already defined a series of local resource permissions which represent the rights to access different system resources. For instance, FilePermission represents the right of accessing (including read, write and execute) to a file or directory. By explicitly specifying suitable FilePermission for the agents from a Mobile Grid Service, Resource providers can prevent those agents from accessing other privacy files which are independent of this service. Other JAAS defined permissions include SocketPermission (access to network) and RuntimePermission (e.g. create class loader, halt JVM).

Besides those JAAS permissions, JADE-S has defined some JADE related permissions which are used in the JADE architecture only. For example, AgentPermission is used to limit the actions such as killing and creating to agents with certain names or owned by certain identities. MessagePermission is used to limit the communication among agents with different owners.

By setting suitable JAAS and JADE permissions in the policy file, we can restrict the agent actions within the execution of Mobile Grid Services in each container (grid nodes). This can prevent Mobile Grid Services from abusing grid resources or inhibiting other Mobile Grid Services' execution.

6.5 Security facilities in MGS API

In this section, we will demonstrate the security support in the MGS API. Since the MGS API is constructed by combining the JADE and Globus toolkits, the security support can also be divided into 2 parts: Globus-side (between clients and Agent Manager) and JADE-side (between Agent Manager and related agents).

Taking the advantages of conforming to the Globus Grid architecture, the Globus-side security in the API can fully employ the facilities in the Globus toolkit. No change is required for the implementation of the components in the MGS API. Moreover, no additional facility for Globus-side security is necessary to be provided in the MGS API.

The situation is different for JADE-side security. The components in the MGS API need to be modified in order to adapt to the security mechanisms provided in JADE-S. Besides, service developer should specifically handle agent implementation and configuration tasks in order to employ the agent security mechanisms. However, normal Grid Services developers are not expected to be familiar with the mobile agent technology and the JADE toolkit. It becomes a barrier for them to use the MGS API to develop secure Mobile Grid Services.

Therefore, extra facilities aiming to help developers to handle the agent implementation and configuration work is provided as the security support in the

MGS API. In the following sections, the details of the security support in the form of MGS component modifications, security libraries and MGS configurations will be described.

6.5.1 Major modifications for MGS components

In this section, the main modifications in the components of the MGS API for adapting to the JADE-S are discussed. The details are as follows:

SecurityHelper object To utilize the functions provided by the JADE-S, a *SecurityHelper* object should be created in each agent. It provides an agent all methods for accessing security functionalities. Therefore, it will be created during the agent initialization of Agent Manager Agent, Task Agent, Monitor Agent and Resource Manager Agent (in the Resource Information Service) in our framework.

However, the *SecurityHelper* object does not implement the *Serializable* interface of Java such that it is unable to move along with the JADE agent during migration. This is owing to JADE-S does not provide full support to agent mobility. Therefore, special handling of the *SecurityHelper* object is required for the Task Agent and the Monitor Agent (which may perform migrations during their execution). Before any migration, the credential will be extracted from the *SecurityHelper* object and then stored in an instance variable in the agent. After that, the *SecurityHelper* object will be discarded. After the move, a new *SecurityHelper* object will be created in the agent again and the extracted credential will be added back to it. This can solve the serialization problem caused by the *SecurityHelper* object and maintain the consistency of the credential in the agent.

Secure message exchange By using JADE-S, signing or encrypting an ACL message can be achieved by invoking the *setUseSignature()* or *setUseEncryption()* method of the *SecurityHelper* object before sending a message. However, sender should obtain public key of receiver first in order to have a successful message encryption. This is achieved by sending principal request to the receiver directly. The *JADEPrincipal* object (contains the public key) of the receiver can be obtained in the reply. This is an essential process for the first time of encrypted message exchange between two agents. Therefore, the handling of principal request is added to all kinds of agent in the MGS Framework. To ease the development of services with secure ACL message exchange, the additional processes for sending secure messages are encapsulated in the security libraries provided in the MGS API (details in section 6.5.2).

For receiving signed or encrypted messages, the checking of the signature validity and the decryption of message is done automatically by JADE-S. Hence, no additional handling needs to be implemented in the API. For the Task Agent (and its subclass – Monitor Agent), message receiving is caused by invoking *receiveMGS()* method. This method will return an ACL message in clear text no matter the received message is signed or encrypted.

Establishment of secure mode Another major modification is the establishment of secure mode. All components of the Mobile Grid Services can be chosen to run in normal mode or secure mode. The additional codes for security support including the MGS security libraries will execute only when the service/agent is in secure mode. This design reduces unnecessary overheads when service providers do not want their services running under security protection. The running mode of a service (Agent Manager) will determine whether the created Task Agents and Monitor Agents run in secure mode or not. It is configured by configuration files.

6.5.2 MGS Security Libraries

The security libraries mainly take charge of the ACL message protection in the Mobile Grid Service (JADE agents communicate and interact with each other by exchanging ACL messages).

The libraries are used in the implementation step of the Mobile Grid Services. Through employing appropriate libraries, the message can be protected against altering and eavesdropping by message signature and message encryption respectively. Therefore, developers can freely decide whether the ACL messages sent by an agent are signed/encrypted or not. The security libraries for each service components' implementation will be described next.

Agent Manager For the Agent Manager component's implementation in a service, Agent Manager should send ACL messages to its Task Agent at appropriate times. In the *AgentManager* class of the MGS API, various methods are provided for sending ACL messages with different protection levels to Task Agent. These methods are *sendACL()*, *sendSignedACL()*, *sendEncryptedACL()* and *sendFullSecureACL()* methods for sending plain text, signed only, encrypted only and both signed and encrypted message respectively. The actual process of signing and encrypting ACL messages (including asking for public keys and principal exchange) is hidden from the developers.

Monitor Agent For the Monitor Agent, the ACL message protection is determined by the *msgNeedSign* and *msgNeedEncrypt* flags in the *MonitorSetting* object. The *MonitorSetting* object is an argument passed in during the creation of the Monitor Agent. These two flags indicate whether the messages sent by the Monitor Agent require signature and encryption respectively. For example, a plain text ACL message will be sent by this Monitor Agent if both flags are set as false. Therefore, developers have to organize suitable settings before passing the *MonitorSetting* object as argument.

Task Agent For the Task Agent, similar to the Monitor Agent, developers are required to pass the *TaskSetting* object with suitable setting as argument during the creation of the Task Agent in order to decide the ACL message protection level. Again, the *msgNeedSign* and *msgNeedEncrypt* flags are available in the *TaskSetting* object for indicating the protection level. The *sendMessage()* method is the main method provided for the Task Agent to send ACL messages according to the two flags in the *TaskSetting* object. By employing the *sendMessage()* method as the unique message-sending method, the ACL messages sent by this Task Agent will have an uniform message protection level.

However, developers sometimes may require various levels of protection for different message exchanges in a Task Agent. Hence, the MGS API provides four extra methods for the Task Agent to send ACL messages with different protection levels. They are *sendClearMsg()*, *sendSignedMsg()*, *sendEncryptedMsg()* and *sendFulSecureMsg()* methods. These four methods will ignore the *msgNeedSign* and *msgNeedEncrypt* flags and send an ACL message in the form of plain text, signed, encrypted and signed plus encrypted respectively. They will be helpful for the developers having special security requirements.

Besides normal message exchange, Task agent will send its task result to Agent Manager by invoking *notifyResult()* or *notifyFinalResult()* method. These methods employ the DF notification in JADE where the result content cannot be protected by encryption. To complement this problem, *notifyConfidentialResult()* and *notifyConfidentialFinalResult()* methods are provided in the API for sending confidential result to Agent Manager. They will set the confidential result in the agent first and then notify Agent Manager through DF service notification. After receiving request for the result, the confidential result will be sent to Agent Manager by encrypted message.

6.5.3 MGS Security Configuration

According to the Mobile Grid Service Framework, a grid supporting Mobile Grid Services must contain a JADE platform (with containers residing on all grid nodes). During each service's initialization, a new service container will be created and connected to the JADE platform. In fact, secure Mobile Grid Services require a secure JADE platform. The secure JADE platform can be realized by using the JADE toolkit with JADE-S add-on and proper configurations.

Therefore, the configuration of MGS involves two parts: JADE platform configuration and service configuration. If the service developers want to employ the security mechanisms (Authentication, authorization, agent permission and message integrity and confidentiality) provided in the Mobile Grid Service Framework, the Mobile Grid Services as well as all JADE containers in the platform must be configured to run in secure mode.

The configuration of a JADE container is done by setting parameters in the JADE configuration file. These parameters determine whether the container is main container, runs in secure mode, registers to which host, etc. For secure mode, an extra JADE policy file describing agent permission is required for each container.

For the service configuration, service providers are required to prepare a file called MGS Configuration file for each Mobile Grid Service. In the MGS Configuration file, service providers can specify the service name, the running mode (normal or secure) and the JADE Configuration file used for service container creation.

MGS Configuration Helper Configuration of each container needs to handle a set of configuration files. We can imagine that configurations of Mobile Grid Services and the grid supporting them involve many configuration files. Consequently, the MGS Configuration Helper is provided to ease the configuration tasks and reduce mistake.

It is a Java program for helping service and resource providers to handle the configuration works of the JADE platform. Service or resource providers can handle the JADE-side configuration works in a graphical user interface of the program instead of dealing with the configuration files separately. (The guide for the configuration work and using the MGS Configuration Helper are presented in Appendix B.)

Chapter 7

Agent Protection for Mobile Grid Services

As the mobile agents in the Mobile Grid Services will migrate to and execute on untrusted hosts, the code or the data of the mobile agents may possibly be modified by the hosts during the execution. Therefore, we should have some measures to protect the agents against malicious hosts which may attack their incoming agents. Unfortunately, this kind of agent protection is not provided in JADE or JADE-S. In this chapter, the mechanism used to achieve the agent protection in the Mobile Grid Service Framework is proposed. Besides, the implementation details of the mechanism as well as its strength and weakness will be discussed.

7.1 Overview

Among several current approaches for agent protection against malicious hosts [30, 31, 32, 33] (including execution tracing, computing with encrypted function, introducing a trusted hardware, and adding time limitation), execution tracing is the most suitable one because of the high feasibility, large scalability and relatively high accuracy. Execution tracing methods employ re-execution to detect malicious actions. For each mobile agent execution in a host, the initial state, inputs from the outer environment and the final state will be recorded in a trace. By using the details in the traces, the execution can be re-executed in another host. If any inconsistent result is found in the re-execution, modification of data or code of the mobile agent by the malicious hosts can be detected.

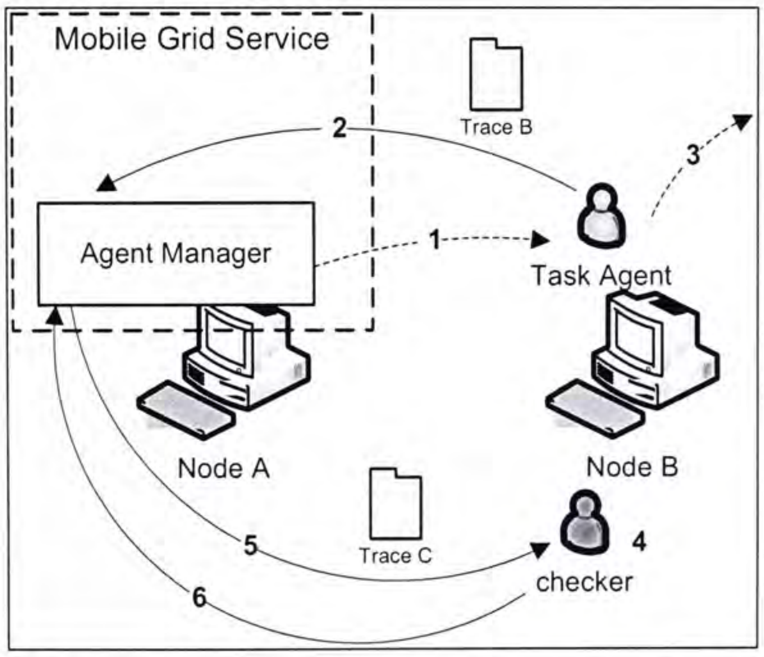


Figure 7-1: The overview of the "Execution Tracing with Randomly-Selected Hosts" mechanism

In our Mobile Grid Service Framework, the agent protection is achieved by combining the "Execution Tracing with Randomly-Selected Hosts" method [34] into the Mobile Grid Service Framework. To gain the protection, service developers are responsible to provide a checker (agent carrying out the re-execution) for each Task Agent. Fig. 7-1 shows the steps involved when the "Execution Tracing with

Randomly-Selected Hosts" method is carried out in our framework. The steps are as follows:

1. The Agent Manager of a Mobile Grid Service at Node A sends its Task Agent to another host (Node B) for execution.
2. After finishing the execution on each host (Node B), the Task Agent will produce a trace (Trace B) for this host and forward it to the Agent Manager.
3. The Task Agent can then migrate to other hosts for other resources and continue its execution.
4. Once the Agent Manager has received the trace from a host (Node B), it will create a checker in that host (Node B).
5. After the Task Agent has visited several different nodes, some checkers will be residing on the visited hosts. The Agent Manager also has received some traces and it will randomly send each received trace to one of those checkers (except the one in the host producing that trace) for re-execution. In this example, Trace C is sent to the checker in Node B.
6. The checker compares its result and the recorded result in the trace after re-execution. If there is any inconsistency, there is an attack. The checking result is positive when consistent re-execution is obtained. Finally, the checker will report the checking result to the Agent Manager.

If the Agent Manager receives any report of attack, it will stop the Task Agent and report it to the client. This can ensure that the mobile agents are not attacked by malicious hosts.

7.2 Major modifications

To apply the "Execution Tracing with Randomly-Selected Hosts" to the Mobile Grid Service Framework, some modifications must be done on the tracing mechanism as well as the MGS Framework. The main modifications are shown in details as follows:

7.2.1 Exempting checking for executions on home host

Since the home host (where the service is setup) is assumed to be trusted, the execution carried out on it should be correct and never suffer attacks. It is wasteful to do any checking for the stages executed on the home host originally. Hence, special handling is designed to reduce those redundant checking. When the Agent Manager receives a trace from a Task Agent who is currently executing on the home host, the trace will not be sent to any Checkers. Instead, this execution stage will be marked as checking completed directly. The treatment will be the same as receiving positive checking result from a virtual checker located on the home host.

This design aims to reduce unnecessary checking and save resources in the grid. Besides, this helps to accelerate the tracing process and allow users to receive the reliable results earlier.

7.2.2 New definition of stage

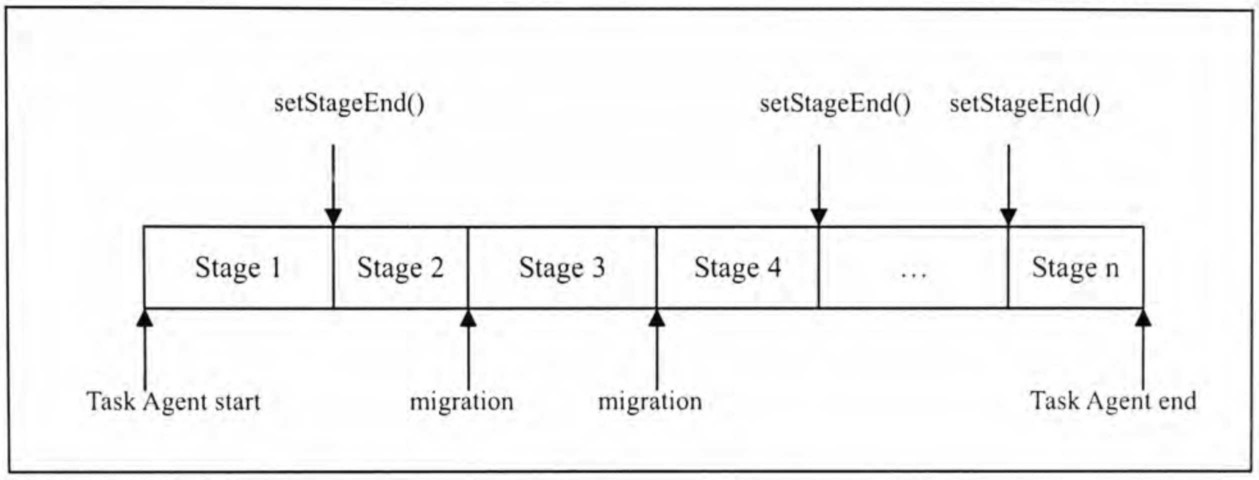


Figure 7-2: Example stage partition in a Task Agent's execution

In the "Execution Tracing with Randomly-Selected Hosts", the execution of the Task Agent is divided into some parts. Each part is called a stage. In the original design, developers have to indicate the stage end before any agent migration. It is done by invoking specific methods in the code of the agent at compile time. However, it is not applicable in our framework where automatic agent migration will occur. Therefore, some modifications are made on the definition of stage.

In our framework, the ending of each stage is identified by user-specified stage end (invocation of *setStageEnd()* method) or migration. Fig. 7-2 is an example showing stage partition in a Task Agent's execution. In the example, the Task Agent enters the first stage after it starts. The first stage will end with the Task Agent running the *setStageEnd()* method. After that, a new stage will start and execute until the migration occurs. In this way, a series of stages will be formed throughout the execution. Finally, the last stage terminates at the end of the Task Agent execution.

This design conforms to the original "Execution Tracing with Randomly-Selected Hosts" such that each stage will be executed on a single host only. In other words, each stage has one and only one execution host. The drawback is that the number of stages in the Task Agent's execution is not under control. Automatic agent migrations may bring uncountable additional stages to the execution and induce significant overheads. However, the number of automatic migrations should be small under normal situation.

7.2.3 Extra operations in Task Agent and Agent Manager

According to the "Execution Tracing with Randomly-Selected Hosts" mechanism, extra operations will be done in Task Agent and Agent Manager during their executions.

For the Task Agent, the extra operations include the creation of trace and the request for commit. At the end of each stage, the Task Agent will create a trace by the details of the initial state of the stage, input ACL messages and the current state. The trace will be sent to the Agent Manager to ask for stage commit. After receiving the reply of Agent Manager, the Task Agent can continue execution or migration. (Details in section 7.3.2)

For the Agent Manager, the additional operations include receiving of traces from the Task Agents, stage commitment, allocation of the received traces to suitable checking hosts and attack handling (presented in next section). It is also responsible to create the Checker and move them to suitable hosts. (Details in section 7.3.1)

7.2.4 Handling of attack

The original "Execution Tracing with Randomly-Selected Hosts" mechanism can discover an attack has possibly occurred during agent execution. However, it is unable to find out the exact host where the attack is located on. In the mechanism, a possible attack is found when the re-execution result is inconsistent with the original execution in certain stage. The origin of the inconsistency may be the interfered original execution (done by Task Agent) or the tampered re-execution (done by Checker in another host). Therefore, the mechanism cannot determine whether the original execution host or the re-execution host is malicious.

Finding out the malicious host in the tracing process is important for our Mobile Grid Service framework. By recognizing the harmful hosts, other Task Agents in the Mobile Grid Service can avoid migrating to these hosts again. The risk of being attacked can be reduced. If we just restart the execution of the Task Agent being attacked without discovering the malicious host, the Task Agent may migrate to the malicious host again and thus attack occurs again in the new execution. The execution may never finish. Marking both suspected hosts as malicious is also impractical because this will waste a reliable resource provider.

To find out the malicious host correctly, at least one more execution for the attack-found stage is required. This step is called judgment in our framework. The location for performing the judgment is important. The original execution host and the re-execution host must not be used as they are suspect. If we choose another visited host, the judgment may be wrong since the chosen host may be malicious. Accuracy of the judgment can be increased by repeating the execution in more hosts. However, the number of available hosts is small in some cases. Besides, this design will generate a large workload to the grid which is not efficient. Hence, the judgment should be done on a trusted host instead.

In our design, the host home is chosen for performing the judgment. The judgment must be correct since the home host is assumed to be trusted. This design can limit the judgment to be finished in one step and speed up the judgment process. Although the judgment will bring extra computations to the home host, the influence should be rare as judgment is required only when attack is detected.

At the beginning of the judgment, the trace for the attack-found stage will be sent to the Checker located on the home host. After the home Checker finishes the re-execution, it will compare the result with that stored in the trace.

Consistent result If the result is consistent, that means the original execution is correct. The original execution is reliable and it can be marked as checking completed. On the other hand, if the re-execution host should be malicious then it will be marked in a blacklist stored in the Agent manager. The blacklist will be sent to all Monitor Agents such that they can then refer to the blacklist when making migration decisions. This can prevent further execution and checking from being carried out on the found malicious hosts.

The checking performed by the malicious Checker cannot be trusted anymore. Any incoming re-execution result from the malicious Checker will be ignored. For the stages whose checking are assigned to be done by the malicious Checker (including those that are waiting for results and those that have already received results), all the checking result will be removed and their traces will be sent to other Checkers for re-execution. This makes sure the checking is reliable.

For the stages whose original executions are done in the malicious host, no extra operation will be done. Even the executions of those stages are done on the malicious host, any incorrect execution result will be found by the checking process. If those stages pass the checking, we can trust their execution is reliable and the overall execution is not affected. This can reduce unnecessary execution caused by the detection of malicious host.

Inconsistent result If the result is inconsistent, that means the original execution is wrong. The execution of the Task Agent is under attacks and its result cannot be trusted anymore. Therefore, the execution of the Task Agent should be stopped and the service requester should be notified for such an attack.

In our MGS framework, the Agent Manager will send messages to stop the Task Agent under attack and its corresponding Monitor Agent (if any). The execution result (stored as a MGSResult object) for the Task Agent will be marked as "attack-found". When the requester of this service obtains the result, he can know the situation and request for the service again (if necessary).

7.3 Implementation details

In this section, the implementation details of all components used for supporting agent protection in the MGS Framework will be presented separately.

7.3.1 Agent Manager

To make the Agent Manager to support agent protection, modifications are made on the *AgentManager* and the *AgentManagerAgent* classes.

For the *AgentManager* class, one of the modifications is the establishment of protection mode. The facilities for agent protection support execute only when the services are running in protection mode. This design reduces unnecessary overheads when service providers do not want their services running under agent protection. Therefore, protected Task Agents can only be created by the Agent Manager under protection mode. At the same time, Task Agents which do not employ agent protection can still be created and run properly when the Agent Manager runs in protection mode.

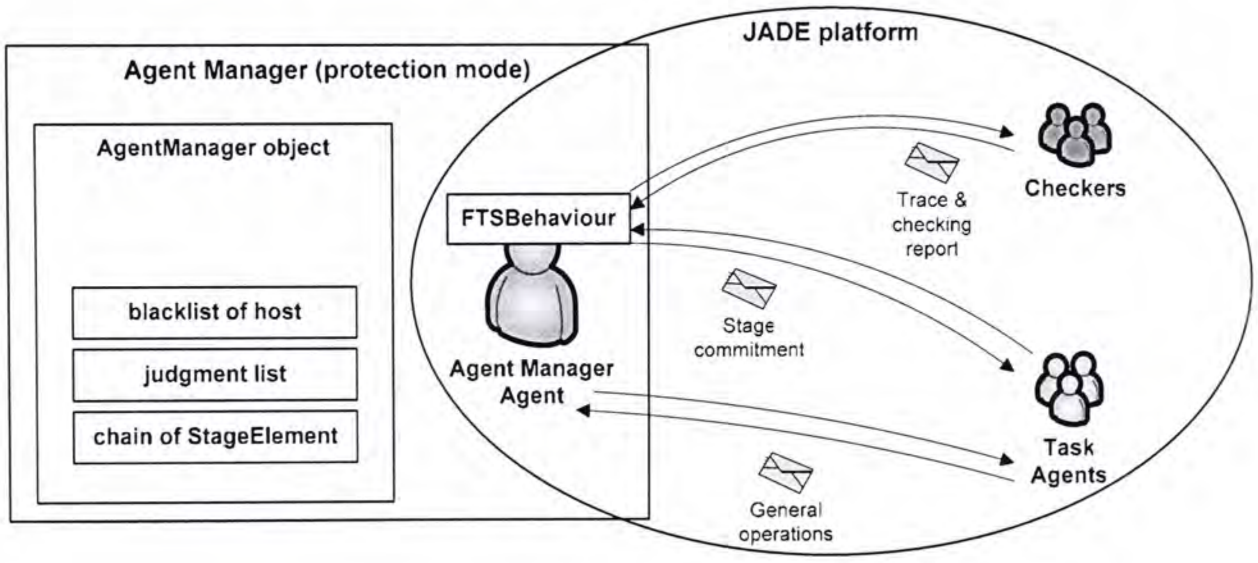


Figure 7-3: Agent Manager in protection mode

Besides the establishment of protection mode, extra lists are added to the *AgentManager* class for storing useful information used by the tracing process (Fig. 7-3). For example, the blacklist of available hosts is used to record all detected malicious hosts in the service; the judgment list is used to store the trace of the stages required to carry out the judgment. For each Task Agent, a chain of *StageElement* objects is stored for keeping track of the traces and other important information for all stages in the whole Task Agent execution. In each *StageElement* object, the trace (contains initial state, final state and all stored messages), the execution host, the assigned checker and the checking result of a stage are recorded. In addition, a list of available Checkers is maintained for each Task Agent such that it can be used for the random Checker selection process.

For the notification of tracing result, the *MGSResult* object is utilized again. After a protected Task Agent finishes its execution, the Agent Manager will mark the received result in an *MGSResult* object. Through the method provided in the *AgentManager* object, the service requester can obtain a "finished" *MGSResult* object even when the checking process has not yet finished. In this case, the *checked* flag will be set to false (indicate that the result is not fully trusted). After all the checking has finished and no attack is found, the *checked* flag will be set to true. If an attack is found, the *MGSResult* object will be set as "finished" and "checked" while its result field will become "ATTACK_FOUND"

For the side of the *AgentManagerAgent* class, a new behaviour class called *FTSBehaviour* is implemented. The *FTSBehaviour* object will be created and executes as a part of the AgentManagerAgent under protection mode. It is responsible for all message handlings involved in the "Execution Tracing with Randomly-Selected Hosts" process. The main operations in the *FTSBehaviour* including stage commitment, Checker selection and receiving re-execution report will be described as follows:

Stage commitment When a protected Task Agent needs to close a stage, it will send a commit request containing the trace and other information to the Agent Manager. In the Agent Manager, the agent's itinerary and the stage number stated in the received commit request will be verified by referring to the chain of *StateElement* object for that Task Agent. Besides, the Agent Manager will check whether the execution host asserted in the commit request is correct. It is achieved by sending message to ask the JADE agent platform directly. After passing all the checking above, the commit request will be accepted and the trace and related information will be stored in a new *StateElement* object. The *StateElement* object will be then added to the tail of the chain.

If the received trace comes from a new execution host which is never visited by the Task Agent, an additional operation will be done in the Agent Manager. A new Checker for the Task Agent should be created and migrated to that host. If there is a Checker created by the same Checker class available on that host already, no redundant Checker will be created. Instead, the existing Checker will be shared for all suitable Task Agents. The AID of the Checker will be then added to the list of available Checkers for that Task Agent.

Checker selection For each received trace, a random Checker selection process will be carried out. In the process, a checker will be chosen randomly from the checker list for each stage not having selected a checker. The choice must obey the rule that the chosen checker is not located on the original execution host for the same stage. If a stage cannot be assigned by a suitable checker, it should wait for the next selection.

After each selection, the traces (containing initial state, final state and all stored messages) for the newly assigned stages will be sent to their chosen Checkers for re-execution. Besides the trace, the AID of the Task Agent will be included. Unless it is in the first stage, the final execution state of the previous stage will be also sent to the Checker. It is used for the checking of consistency between previous final state and current initial state.

Receiving re-execution report After the Checkers finish their checking, the Agent Manager will receive the reports from them. For the positive checking results, the checking result field in the *StageElement* objects representing the appropriate stages will be marked as true. The checking of those stages is finished. For negative checking results, judgments are required to determine the malicious hosts. The traces and other information of those stages will be sent to the Checker at the home host for re-execution. Since the judgments must be done one by one, the judgments not yet finished will be stored in the judgment list in the *AgentManager* object. After checking results are received from the home Checker, the results will be matched with corresponding judgment cases stored in the judgment list. The checking result field in the corresponding *StageElement* objects will be set as true if the original execution is correct. If opposite, they will be set as false. The detected malicious hosts will be handled further (refer to section 7.2.4).

In fact, the Agent Manager provides three additional methods which are particularly prepared for the agent protection support. The three methods are:

void createProtectedAgent(String agentName, String agentClass, String checkerClass, Object[] agentArgs)

This method is used to create a new "protected" Task Agent in the service. It is similar to the *createAgent* method except the created Task Agent will be executed under agent protection support. The argument *checkerClass* is used to specifiy the Java class for the corresponding Checker of the created Task Agent. Checker will not be created during the method invocation. Instead, it will be created after the Agent Manager receives a valid trace from that Task Agent.

void createProtectedAgentWithMonitor(String agentName, String agentClass, String checkerClass, Object[] agentArgs, MonitorSetting setting)

This method is similar to the one above but it will create a default protected Monitor Agent at the same time. The argument setting is used to configure the default Monitor Agent.

String checkingAtHome(String agentName)

This method is designed for ensuring that the tracing process of the specific Task Agent (specified by the argument *agentName*) can be completed. In fact, the tracing process may be unable to end in some cases. For example, if a Task Agent has visited too few different hosts throughout its execution, some stages of the execution may have no appropriate Checker for performing re-execution. The result is that the tracing cannot be finished. In this situation, the *checkAtHome()* method should be invoked. After that, a suitable Checker for that Task Agent will be created at the home host. All the unchecked stages will be assigned by the home Checker. Thus, all remaining re-execution can be done on the home host and the tracing can be finished.

7.3.2 Task Agent

The protected version of Task Agent supporting agent protection is quite different from the normal version. The protected version of Task Agent is composed of three classes. They are *ProtectedTaskAgent*, *FTSLayer* and *ProtectedTaskBehaviour*. Unlike the flexibility of the Task Agent Template in MGS API, some restrictions are added such that it can run properly in the protection mode. For example, only one behaviour (ProtectedTaskBehaviour) will be added to the agent.

ProtectedTaskAgent class It is the main body of the protected Task Agent. It extends the *TaskAgent* class in the MGS API to inherit the basic functions of Task Agent. In addition, it will instantiate a *FTSLayer* object which provides the essential functions for the "Execution Tracing with Randomly-Selected Hosts" mechanism. Moreover, it will create a *ProtectedTaskBehaviour* object for doing the service task.

Some methods provided in the *TaskAgent* class are overridden in the *ProtectedTaskAgent* class. For example, *moveMGS()* in Task Agent only starts the migration process; in protected Task Agent, *moveMGS()* will carry out the commitment process (send trace and request for commit) before migration. Besides, new *receiveMGS()* method will not only receive incoming ACL messages, but also store the received messages to the message log.

To implement a protected Task Agent, developers are required to extend the *ProtectedTaskAgent* class. The class contains six abstract methods which must be implemented by the developers. They are:

- public abstract void normalExecution()

  The program logic of the service tasks should be implemented in this method.

- protected abstract void msgHandler()

  This method will be invoked for each iteration. The handling of any incoming ACL messages should be implemented. Template is provided where handlings of all essential messages for MGS and agent protection are implemented. Developers only need to manage the user-defined messages.

- protected abstract boolean endChecking()

  This method is used to determine whether the execution of the Task Agent is complete. It should return true when stopping criteria is met.

- protected abstract byte[] getInitState()

  This method is responsible for extracting the initial execution state from specific instance variables.

- protected abstract byte[] getFinalState()

  This method is responsible for extracting current execution state from the core instance variables.

- protected abstract void updateInitState()

  This method is responsible for updating the specific instance variables (used for extracting initial state) by the current values of the core variables. It will be invoked when a new stage starts. Since every Task Agent will have their own structures of execution state, this method is defined as abstract such that developers must implement it.

ProtectedTaskBehaviour It is the class dealing with the actual service task. The abstract methods *msgHandler()*, *normalExecution()* and *updateInitState()* in the

ProtectedTaskAgent class will be invoked by this class.

Figure 7-4 shows the core pseudocodes for the *action()* method in the *ProtectedTaskBehaviour* class. At the beginning of each invocation of *action()*, *msgHandler()* will be called to receive one incoming message(if any) and try to react with it (in line 2). After that, *normalExecution()* is invoked if the agent does not run in protection mode (in line 17). Otherwise, the execution of each stage is divided into three phases: begin phase, execution phase and end phase. The begin phase represents the beginning of a stage. The *beginStage()* method in the *FTSLayer* object and the *updateInitState()* method are invoked to update the initial state in the Task Agent and start a new stage respectively (in lines 5 to 6).

After that, the execution phase is entered. The execution of the service task is performed in this phase. The *normalExecution()* will be invoked only during this phase (in line 10). When the stage is going to end, the end phase is entered. The *commit()* method in the *FTSLayer* object is called for carrying out the commit process (in line 13). The initial and final states obtained by invoking *getInitState()* and *getFinalState()* method will be passed as arguments of the *commit()* method. In the method, a trace will be created and sent to the Agent Manager for requesting commit of the stage. After the commit is accepted, the start phase will be entered again.

After the *action()* method returns, the *endChecking()* method (defined in the *ProtectedTaskAgent* class) will be invoked for determining whether the task execution should be finished. The *action()* methods will be invoked repeatedly until the *endChecking()* returns true (i.e. stopping criteria is met).

```
void action(){
    msgHandler();
    if(agent is in protection mode){
        if (begin phase){
            ftsLayer.beginStage();
            updateInitState();
            enter execution phase;
        }
        else if (execution phase){
            normalExecution();
        }
        if (end phase){
            ftsLayer.commit(getInitState(),
            enter begin phase;
        }
    else{
        normalExecution();
    }
}
```

Figure 7-4: The core pseudocodes for the ProtectedTaskBehaviour class

FTSLayer class This class is responsible for providing methods to allow the Task Agent to perform the "Execution Tracing with Randomly-Selected Hosts" mechanism. Besides, it keeps track of the agent's itinerary (execution locations for each stage) and the current stage number during the execution of the Task Agent. This class is also used to store the information of the home host and the Agent Manager for the use of tracing communication.

In order to carry out accurate re-execution in the checking, all message inputs of each stage must be replicated completely during the re-execution. A message log (implemented in form of the *MessageLog* class) residing in the *FTSLayer* object is for this purpose. The *msgHandler()* method will be invoked every iteration. In the method, it will try to receive one ACL message from the incoming message queue. If the queue is empty, null message will be received. For each iteration of the task execution, the received message including related message, unrelated message and null message will be recorded in the message log.

In the *MessageLog* object, unrelated messages will be stored as null messages in the message log in order to save the storage space. To further reduce the size of the message log, only non-null messages and the iteration number of their occurrences are recorded. In this way, the huge amount of null messages (since most of the iterations should have no incoming message) can be kept away from the message log and the size of the message log can be minimized.

The most important operation provided by *FTSLayer* class is the *commit()* method. It is used to create the trace and send commit request to the Agent Manager at the end of each stage. The trace contains the initial state and the final state (in form of byte arrays) as well as the message log of the current stage. Besides, the current stage number and the agent's itinerary will be included in the commit request and sent to the Agent Manager for the trace consistency checking. The method will return only after it receives the acceptance of the commit request from the Agent Manager. This makes sure that new stage starts only after the previous stage ends.

7.3.3 Monitor Agent

Since the *MonitorAgent* class (the basis of the Configurable Monitor Agent) does not extend the *ProtectedTaskAgent* class, it does not support agent protection. On the other hand, the *MonitorAgent* class is implemented by extending the *SimpleTaskAgent* class. By using the Convert Tool provided in the MGS API, the classes for the protected version of Monitor Agent as well as the corresponding Checker can be generated. When the *createProtectedAgentWithMonitor()* method is invoked, a protected Monitor Agent will be created using the *ProtectedMonitorAgent* class (with the specified protected Task Agent).

The protected Monitor Agent executes in the same way as the normal Monitor Agent does except it will also act as a protected Task Agent to carry out the operations for agent protection (e.g. sending trace to the Agent Manager). The execution state of the protected Monitor Agent includes the stored details of the best chosen host as well as the CPU, RAM and HD values of each available host. Without protection on the Monitor Agent, attack over Monitor Agent cannot be detected and it may lead to improper execution of the Task Agent (e.g. wrongly send "move" message to Task Agent). For the tracing result, service requester can obtain the *MGSResult* object for the specified Monitor Agent (as a Task Agent) in order to check the checking result. This provides a complete agent protection covering both the Task Agent and Monitor Agent.

7.3.4 Checker

To employ the agent protection provided in the MGS Framework, developers are required to prepare a Checker for each protected Task Agent. Checker is used to re-execute the tasks done by the protected Task Agent in order to make sure that the result produced by the Task Agent is reliable. The re-execution done in the Checker should be exactly the same as the original execution completed in the Task Agent such that their results will be identical. In fact, simplification of the re-execution is allowed. For example, re-execution can only concentrate on a single instance variable and execute the codes which will affect the final value of that variable. However, the number of iterations (e.g. cannot check first 10 iterations only) and messages received (e.g. cannot remove any *receiveMGS()* operations) in the re-execution must be kept unchanged.

Each Checker is created at the home host by the Agent Manager at the appropriate time. It will move to a suitable container immediately after the creation. When the Checker receives a trace from the Agent Manager, it will start re-execution according to the traces. After the re-execution, the Checker will report the consistency of the execution results to the Agent Manager and start re-execution for the next trace.

The structure of the Checker is similar to that of the protected Task Agent. The Checker is also composed of three classes. They are *Checker*, *FTSLayerRE* and *CheckingBehaviour*.

Checker class It is the main body of the Checker. It will instantiate a *FTSLayerRE* object which provides the essential functions for the Checker to perform the "Execution Tracing with Randomly-Selected Hosts" mechanism. Moreover, it will create a *CheckingBehaviour* object for doing the re-execution.

To implement a Checker, developers are required to extend the *Checker* class. The class contains six abstract methods which must be implemented by the developers. They are:

- public abstract void normalExecution()

  The program logic of the corresponding service tasks should be implemented in this method such that re-execution can be done during checking.

- protected abstract void msgHandler()

This method will be invoked for each iteration. The handling of any incoming ACL messages should be implemented same as those of the corresponding Task Agent. Unlike the one in the TaskAgent class, this method will obtain the incoming messages from the message log stored in the trace.

- protected abstract boolean endChecking()

  This method is used to determine whether the execution of the task is complete. It should be equal to the same method in the Task Agent.

- protected abstract byte[] getFinalState()

  This method is responsible for extracting current execution state from the core instance variables. It should be equal to the same method in the Task Agent.

- protected abstract void restoreState(byte[] state)

  The method will be used to restore the instance variables in the Checker to the condition recorded in the specified execution state.

- public abstract boolean stateEquals(byte[] state1, byte[] state2)

  This method is used to determine whether two execution states are equal. Since execution state for each Task Agent should be different, developer should provide method to check the equality of any two states. It should return true if the two states are equal.

All methods provided in the *TaskAgent* class will be also available in the Checker class. Therefore, developers can implement their Checkers by simply copying the codes in the *normalExecution()* of the *TaskAgent* class to those of the Checker class. However, to eliminate the influence of the re-execution over other agent and simplify the work of Checker, those methods involve sending ACL messages (e.g. *sendMessage()* and *notifyResult()*) are dummy. The result is that no real ACL message will be sent during the re-execution.

CheckingBehaviour class It is the class simulating the actual service task. The abstract methods *msgHandler()*, *normalExecution()* and *restoreState()* in the *ProtectedTaskAgent* class will be invoked by this class.

Figure 7-5 shows the core pseudocodes for the *action()* method in the *CheckingBehaviour* class. At the beginning of each invocation of *action()*, *realMsgHandler()* will be called to receive one real message and try to react with it (in line 2). The potential received messages are the trace and the "end" message from the Agent Manager. The received traces will be stored in a trace list. This allows the Checker to receive other traces even when it is performing re-execution for a trace. After that, *msgHandler()* is invoked (in line 3). The *msgHandler()* method will get the message from the message log instead of the real incoming message queue.

The checking process is also divided into three phases: begin phase, execution phase and end phase. The begin phase represents the beginning of a stage. At the beginning of the begin phase, the first trace in the trace list will be taken out. The trace will be checked if it is valid before the re-execution (in line 6). The validity of the trace is determined by the consistency of the previous final state and the current initial state stored in the trace. If the trace is not valid, negative checking result will be sent to the Agent Manager directly by invoking the *report()* method (in line 8). Then, the *action()* method will return. If the trace is valid, it will be used to initialize the *FTSLayerRE* object and restore the instance variables in the Checker to the condition recorded in the initial execution state (in lines 11 to 12).

```
1    void action(){
2         realMsgHandler();
3         msgHandler();
4         if (begin phase){
5              if (traceList is not empty){
6                   check if the 1st trace the list is valid;
7                   if(trace not valid) {
8                        report(false);
9                        return;
10                  }
11                  use the trace to restore checker's state;
12                  ftsLayerRE.beginStage();
13                  enter execution phase;
14             }
15        }
16        else if (execution phase){
17             normalExecution();
18        }
19        if (end phase){
20             report(ftsLayer.REcommit());
21             enter begin phase;
22        }
23   }
```

Figure 7-5: The core pseudocodes for the CheckingBehaviour class

After that, the execution phase is entered. The re-execution of the service task is performed in this phase by repeatedly invoking the *normalExecution()*(in line 17). When the stage is going to end (i.e. migration or user-specified stage end occurs), the end phase is entered. Then, the *commit()* method in the *FTSLayerRE* object is called for comparing the final states of original execution and re-execution. The result will be sent to the Agent Manager (in line 20). After that, the start phase will be entered again. Next checking will start when the trace list is nonempty. The execution of the Checker will never end unless "end" message is received from

Agent Manager.

FTSLayerRE class This class is responsible for providing methods to allow the Checker to perform the "Execution Tracing with Randomly-Selected Hosts" mechanism. At the beginning of each re-execution for a trace, it will store all the information such as final state and message log recorded in that trace. Besides, this class is used to store the information of the home host and the Agent Manager for the use of tracing communication.

The most important operation provided by *FTSLayerRE* class is the *commit()* method. The method will be used at the end of the re-execution. It is different from the one in the *FTSLayer* object for the Task Agent which is responsible to create trace and send commit request to the Agent Manager. The *commit()* method in the *FTSLayerRE* object will compare the current state of the re-execution (at the end of the stage) and the stored final state of the original execution. The comparison is done by invoking the *stateEquals()* method in the *Checker* class. It will return true if the two execution states are consistent.

The stored message log is important to simulate the execution environment of the original execution. During re-execution, the *msgHandler()* method will get the message from this message log instead of the real incoming message queue in the Checker. The well-ordered messages in the message log will be obtained consecutively during the re-execution in the Checker. Owing to the special design of the protected Task Agent's structure and the management of the message log, the same message will be obtained and handled at the same iteration during the original execution and the re-execution. As a result, we can completely simulate the execution (including receiving of message) during the re-execution process.

Even for a stage which is ended by an unpredictable automatic migration during the original execution, the Checker can recognize the end of the stage when it obtains a "move" message (sent by the Monitor Agent) from the message log at certain iteration. Then, it can stop the re-execution and start the comparison of the result.

7.4 Discussions

The performance of "Execution Tracing with Randomly-Selected Hosts" is evaluated in [34]. In this section, the strength of the agent protection in the MGS Framework is demonstrated by analyzing different kinds of attacks. Moreover, the weakness and the major overheads of the proposed mechanism will also be discussed.

7.4.1 Against modification of code and data

Modifications of the code or data can be detected. Consider the case that a malicious host modifies the code and data of the Task Agent in a service. If the code and data for re-execution and execution are different, the resulting state of the re-execution should be different from the recorded final state in the trace. Attack can be detected when the Checker compares the two states. If the modification does not affect the result, it can be ignored since no damage is caused.

7.4.2 Against masquerade

Since agent protection is provided when the Mobile Grid Service is under secure mode, the identities of the hosts and agents are under protection. It is impossible to forge an agent's identity under the protection of authentication. Therefore, masquerade is impossible.

7.4.3 Against fake information in trace

Fake information in traces may prevent the tracing from detecting attacks. Since the traces are created by the Task Agents executed in untrusted hosts, the information may be false when the agents are under attacks. In our framework, it is prevented by checking the validity of the received trace before re-execution. By comparing with the final state in the previous trace, fake initial state in a trace can be detected. On the other hand, fake final state will be discovered as an attack after the re-execution. Furthermore, the execution host marked in the trace is verified by querying the JADE platform.

7.4.4 Against escape from re-execution

Any stage is impossible to hide from the whole agent execution. The validity of the received trace is checked before re-execution by verifying the stage number and comparing the previous final state and the current initial state stored in the trace. Any missing of trace will be detected, thus no stage can escape from re-execution.

7.4.5 Against collaboration of different hosts

The hosts assigned for re-execution are unpredictable. They are all randomly selected by the Agent Manager which is located on a trusted host (the home host). Moreover, the traces contain no information about their original execution hosts. Checkers are unable to know the identities of the original hosts during the checking process. Hence, the difficulties for several hosts to collaborate are increased.

7.4.6 Detection of malicious host

Owing to additional judgments conducted at trusted hosts (the home host of the services), the malicious hosts performing the detected attacks can be identified. This can decrease the risk of attack over later executions.

7.4.7 Weaknesses

This method has the common weakness for execution tracing approaches. Although attacks over agents can be detected, they cannot be prevented at all. Moreover, the detection of attacks cannot be achieved immediately. The tracing process can only be started after the Task Agent has visited enough hosts. The reason is that the approach needs a certain number of visited hosts for random selection. Furthermore, it may not be appropriate for any types of applications. For the applications requesting a lot of data from their hosts, this approach may bring significant overheads to them due to the transfers of the traces over the network.

Chapter 8

Performance Evaluation

Experiments were conducted to evaluate the performance of Mobile Grid Services developed by using the MGS API. In the experiments, the execution times of different Mobile Grid Services were executed under different settings. From their execution times, we can examine the performance and the overheads of the Mobile Grid Services. The details, results and analysis of the experiments will be presented in the following sections.

8.1 Experimental Setup

The experiments were carried out in a grid with four nodes running Fedora Core 3. Each node is equipped with 3.2GHz dual-Xeon CPU and 2 GB memory. The machines were connected through a dedicated 10Mbit/sec switched Ethernet. They were all running with Globus toolkit 4.0 [8] and JADE 3.4. Their Java Virtual Machine versions were JDK5.0.

In the experiments, the four nodes were called Host A, B, C and D. Host A acted as the home host where the experimental services were running on it. On Host B, the JADE main container and the Resource Information Service were running. The roles of Host C and Host D were providing available platforms for agents to migrate to.

8.2 MGS Performance

8.2.1 Experiment details

In this experiment, the load balancing performance of MGS in different number of available hosts is measured.

To test the performance of the Mobile Grid Services, a sorting service was implemented by using the MGS API. When the service was requested, a new Task Agent would be created. Monitor Agent was created with the Task Agent in order to handle the resource information and make migration decision. After that, 600000 integers were generated randomly and insertion sorting was carried out in the Task Agent. After it finished the sorting, it would notify the Agent Manager of the service. As the service was a Mobile Grid Service, the Task Agent could migrate to other hosts with better resources during the sorting process.

The sorting service was setup on Host A in normal mode (no security facilities running) first. To test the load balancing, the sorting service was requested 50 times simultaneously with various numbers of hosts (1, 2, 3 and 4) available in the grid. Their execution times were measured as the time from the Task Agent creation to the receiving of Task Agent's final notification.

The experiment was repeated by requesting the service for 10 times and 1 time. To obtain more reliable results, 2, 10 and 5 trials were performed respectively in the experiments simultaneous requesting service for 50, 10 and 1 times. The average execution times of the sorting service under different settings are shown in Table 8-1 and Figure 8-1.

8.2.2 Experiment results

Table 8-1: AVERAGE EXECUTION TIMES (IN SECONDS) OF THE SORTING SERVICE UNDER DIFFERENT SETTINGS

	1 host	2 hosts	3 hosts	4 hosts
50 services	11645.0	4571.3	2415.9	1772.0
10 services	1939.6	962.9	717.6	691.2
1 service	266.6	266.5	266.6	267.2

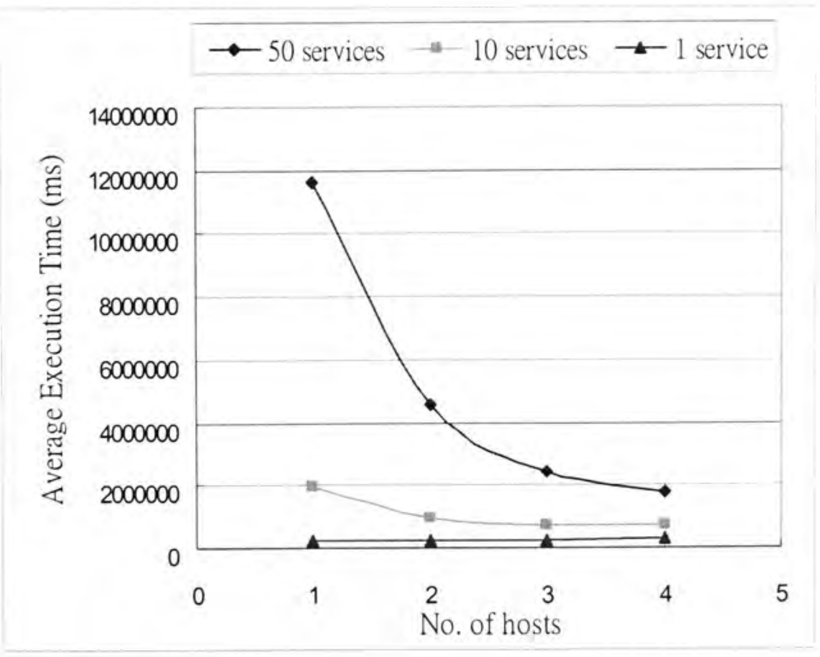


Figure 8-1: The changing of the execution time against the number of available hosts

For invoking 50 sorting services at the same time, the host was overloaded heavily due to work load of services when one host existed only. The created Task Agents in the services needed to share the CPU cycles and perform the execution by turns. The result was that the average execution time of each service was over 11000 seconds (which was 43 times more than the single service's execution time). When one more host was available, the average execution time reduced to about 4600 seconds which was 39.26% of the single host's execution time. The work load of the tasks was shared between the two hosts in the grid and the service performance improved significantly. When there were three hosts, the execution time further improved to 20.75% of the single host's execution time. For using four hosts, the average execution time reduced to 15.22%. The curve was similar to the curve "y=1/x" and we can see that the execution time tended to be further decrease when more hosts are available. This shows that the overloading problem can be solved and the performance of the service can be improved by providing more hosts and resources for service migration.

When 10 sorting services executed together in home host only, overloading occurred and average execution time was nearly 2000 seconds (over 7 times of the single service's execution time). When the grid size increased to 2, 3, and 4, the average execution time improved to 49.65%, 37.00%, and 35.63% of execution time in a single host respectively. We can observe that the improvement was minor when available hosts changed from 3 to 4. Their execution times appeared to approach a boundary.

The reason is that all Task Agents were created on the home host (Host A) and they would initially execute at there. At this part of execution, the Task Agents needed to compete for resources (mainly CPU cycles) in the overloaded home host before they had opportunities to migrate. Owing to the design and the configuration of the Resource Information Service and the Monitor Agents, the formation of the load balancing required some time to finish. Thus, each agent spent certain time on the former part of the service execution but only little work could be done due to the overloading. When several hosts were available (i.e. the execution time became short), the overall execution times of the Task Agents were mainly contributed by this overloading period. Further adding available hosts in the grid could not speed up the load balancing and so could not shorten the length of this period. Therefore, the execution time of the services could not improve significantly when approaching the boundary even if more hosts are added.

For requesting one service only, the average execution time was maintained at around 267 seconds when the size of the grid changed from 1 to 4. This phenomenon can be explained by the fact that the home host's resources being consumed by a single service was little such that agent migration was not triggered. Therefore, all the measured execution times were similar which represented the time required for a Task Agent to execute at home host until it ended.

From Figure 8-1, we can find that the curve for 50 services is the steepest among the three curves and it tends to be flattening slower. This shows that load balancing performs better when the home host is more overloaded. It is reasonable because a higher level of overloading will lead to a longer execution time which can lessen the effect of boundary execution time caused by the gradual load balancing formation.

8.2.3 Discussion

From this experiment, we can see that Mobile Grid Services are able to relieve the overloading problem and make use of idle resources in the grid by employing their migration ability. In fact, the testing service used in this experiment is implemented in a general way. We can improve the service performance by specifically modifying the service in the implementation stage. By proper Task Agent implementation, the Task Agents in the service can be moved to other available hosts randomly at the beginning of their execution. This can avoid too many Task Agents (created by multiple service requests) working on a particular host initially. This procedure is suitable for the services which are expected to be requested heavily within a short period (i.e. a large number of Task Agents are created simultaneously).

In the MGS Framework, the load balancing is achieved by appropriate Task Agents' migrations (which are decided by their corresponding Monitor Agents individually). Developers can configure or overwrite the migration decision policies in the Monitor Agents in order to improve the load balancing among the available hosts.

For example, instead of deciding to move to the host with highest resource values, developers can adjust the Monitor Agents to find several hosts with higher resource values and decide to migrate to one of them randomly. This allows the load share to the available hosts faster and the balancing to be achieved in a faster way.

If the loads of the hosts are unbalance, the resource data will reflect this fact and Monitor Agents in more busy hosts will decide to move to less busy hosts. Finally, load balancing should be achieved unless the loads naturally cannot be uniformly distributed (e.g. existence of an especially heavy task). To relieve that problem, one solution is modifying the migration decision policy such that moving to a visited host is discouraged (e.g. deduct the resource values of visited hosts). This can prevent the unnecessary non-stop migration due to naturally unbalance loads. When there is a change on the load (e.g. new tasks appear and old tasks finish), migration process will be carried out again in order to maintain the load balancing.

8.3 MGS Overheads

8.3.1 Experiment details

A set of new Mobile Grid Services was used in this experiment. This experiment concentrated on the general overheads, migration overheads and message overheads of Mobile Grid Services in different modes.

A new service called Service X was implemented by using the MGS API. When the Service X was requested, a new Task Agent would be created. Monitor Agent would not be created such that no unexpected migration would occur to influence the result. At home host (Host A), the Task Agent would carry out 100000 times the prepared task. For each prepared task execution, 200 random integers were generated for doing summations with a variable and other 200 random integers were generated for subtractions. Meanwhile, it would send and receive five clear ACL messages to and from an agent located in Host B. After that, the Task Agent would migrate to Host C and then Host D to carry out 100000 times the prepared task respectively. Finally, it would move back to Host A.

In the experiment, Service X setting up on Host A in normal mode was invoked.

The time started from the Task Agent initialization to the end of execution was measured as the execution time of the service.

Service X was modified to a set of new services for measuring the execution times of the service under different settings. The new services were equal to Service X except for the specific differences below:

1. No migration.
2. No ACL message sending and receiving. (Include versions with and without migration)
3. Running in secure mode. (Include versions with and without migration)
4. Using signed message instead of clear message. (Include versions with and without migration)
5. Using encrypted message. (Include versions with and without migration)
6. The prepared task was changed from 200 times to 100 times of random integer summations and subtractions.
7. Service developed in standard Grid Service (GS) instead of Mobile Grid Service (MGS). The migration and message transmission were missing due to the absence of related supports.
8. Standard Grid Service version. The prepared task was changed from 200 times to 100 times of random integer summations and subtractions.

All the new services were executed separately and their execution times were recorded. For each service and setting above, 10 trials were performed. The average results of the experiments were shown in Table 8-2.

8.3.2 Experiment results

Table 8-2: THE EXECUTION TIMES (IN MILLISECONDS) OF Service X UNDER DIFFERENT SETTINGS

Service form	Running mode	Message mode	100 summations and subtractions without migration	200 summations and subtractions without migration	200 summations and subtractions with 3 migrations
GS	N/A	None	4053.5	7991.0	N/A
MGS	Normal	None	5038.3	8892.4	9867.3
MGS	Normal	Clear	N/A	9019.9	10030.5
MGS	Secure	Clear	N/A	9027.6	10207.7
MGS	Secure	Signed	N/A	9072.2	10420.5
MGS	Secure	Encrypted	N/A	10734.9	11922.1

In Figure 8-2, the execution times of the service without message transfer in the form of standard Grid Services (GS) and Mobile Grid Services (MGS) are compared. For both the prepared task consisting of 100 or 200 summations and subtractions, the MGS version requires a longer time to finish. This is reasonable as MGS has a general overhead caused by the agent creation and the additional works in agent execution (e.g. examining incoming messages and checking if stopping criteria are met). For 100 addition and subtraction, the MGS version executed 984.8 milliseconds (24.30%) longer than the standard Grid Service version. For 200 addition and subtraction, the MGS version executed 901.4 milliseconds (11.28%) longer than the standard Grid Service version.

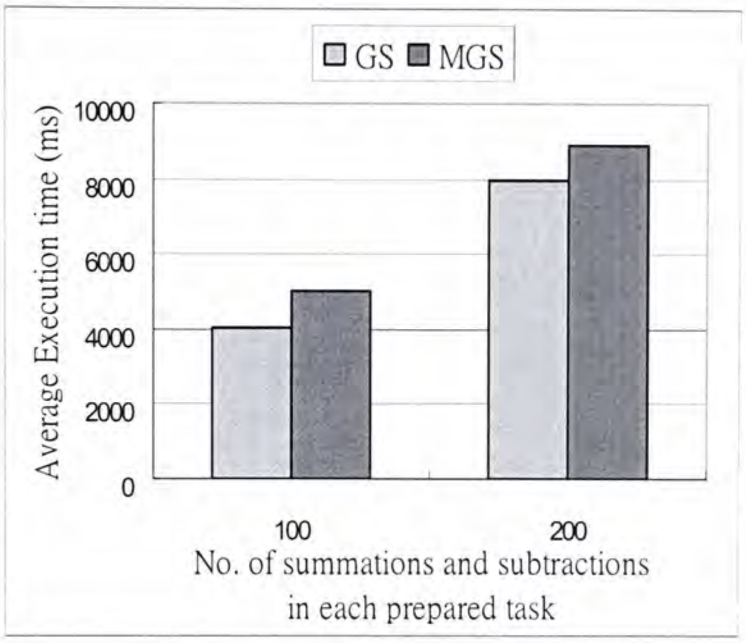


Figure 8-2: Execution times of Service X in form of standard Grid Service (standard GS) and Mobile Grid Service (MGS)

We can find that the MGS overheads are similar for different prepared task contents. Actually, the MGS overheads mainly contributed by the essential works in agent execution such as examining incoming messages and they are independent from the prepared task in each iteration. By taking the average value of the execution time's differences, we can estimate that the general overhead of Mobile Grid Services is about 0.003 milliseconds per iteration. For example, the overheads will contribute to 0.3% of the overall execution time if the task spends 1 millisecond in each iteration.

From the experimental results, we can also see that the task execution time in each iteration determines the proportion of the MGS general overheads in the overall execution. Longer execution in each iteration will make the MGS overheads less significant.

Figure 8-3 shows the migration overheads of the Mobile Grid Services under different running modes and message forms. The migration costs represent the overheads of each agent migration in the Mobile Grid Services. They are calculated by one-third of the execution time differences between service with migration and without migration under the same setting (since three migrations were involved).

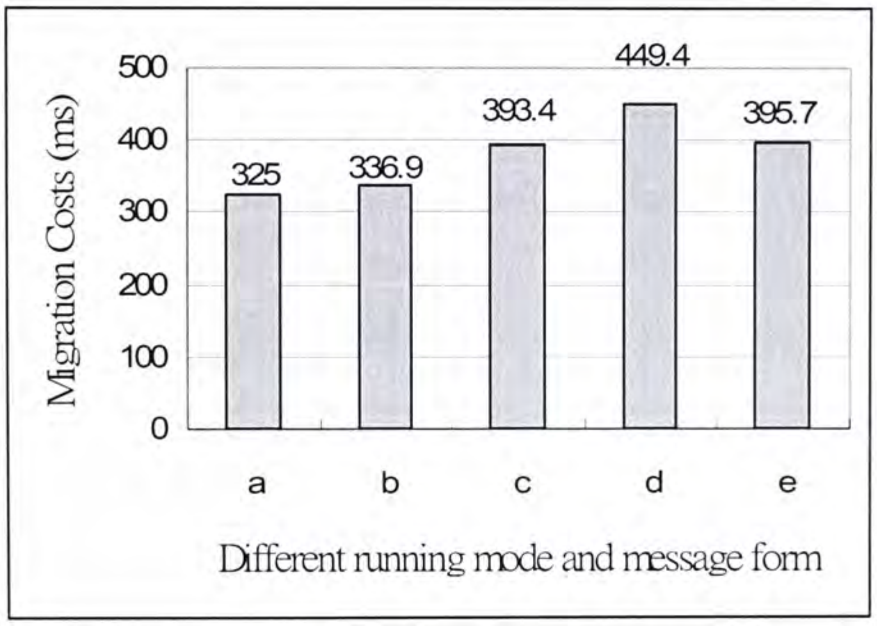


Figure 8-3: The migration costs under different running modes and message forms, where a = normal mode without message transmission; b = normal mode with clear message transmission; c = secure mode with clear message transmission; d = secure mode with signed message transmission; e = secure mode with encrypted message transmission

From the graph, we can observe that the migration costs are alike under different settings. By taking the average value, the overhead for each migration is about 330 milliseconds in normal mode while 410 milliseconds in secure mode. Migration introduces more overheads in secure mode than normal mode because extra secure process and checking is present such as authentication and agent permission checking. In fact, the migration cost will be affected by the size of the agent execution state. For example, an agent with a lot of instance variables should have higher migration overheads.

Figure 8-4 shows the estimated message overheads in different running modes and message forms. The values are calculated by one-fifth of the execution time differences between the service without ACL messages transmission in normal mode and services under other settings (since five message sending and receiving were involved). The values stand for the costs of sending and receiving one ACL message under different modes.

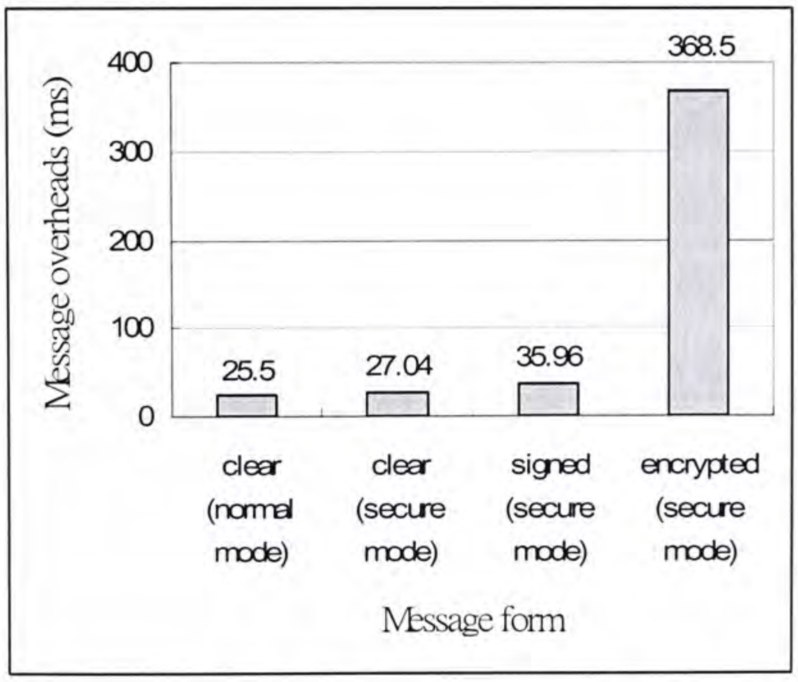


Figure 8-4: Message overheads for different message forms and running modes

From Figure 8-4, we can see that secure mode does not bring significant overheads to the service handling clear ACL messages. The costs for clear message are very similar in normal and secure mode. Both values are less than 30 milliseconds and this shows that handling clear ACL messages is a light-weight operation. The overhead for signed messages (35.96 milliseconds) is higher but it is still not a heavy operation. Comparatively, the handling of encrypted messages introduces a much larger overhead. The cost for an encrypted message is 10 times higher than that for a signed message. However, it is reasonable as encryption and decryption are known as time-consuming processes. In fact, the message overheads above will be affected by the size of the ACL messages.

8.3.3 Discussions

From this experiment, we know the general overheads, migration overheads and the message overheads of the Mobile Grid Services. Although those overheads cannot be completely eliminated, service developers can maintain good service performance by appropriate Task Agent implementation (e.g. reduce the number of iterations, migration and message transmission in the Task Agent).

In the MGS Framework, we have tried to diminish the influence of the migration overheads by minimizing number of migration in the automatic load balancing process. Unnecessary migrations are avoided by the "Gain" checking in the migration decision policy of the default Monitor Agent. This checking makes sure that the resource values (i.e. CPU, RAM and HD) of the decided host have a significant improvement (more than 5%) when compare with the current host. Otherwise, migration will not take place.

On the other hand, only part of Monitor Agents will receive resource data in each round of data distribution. This can prevent all Task Agents from migrating to the currently best host together and overloading that hosts. Unnecessary migrations due to this kind of collective migration can be avoided.

8.4 Agent Protection Overheads

8.4.1 Experiment details

In this experiment, the overheads of the Mobile Grid Services using agent protection are considered. For the Mobile Grid Services, agent protection is provided by the "Execution Tracing with Randomly-Selected Hosts" [34] mechanism. In the mechanism, the execution of Task Agent is divided into some stages and extra operations are required at the end of the stages. The main purpose of those operations is to produce traces which store partial execution results. The traces will be used by Checkers to carry out re-execution and find out any inconsistent results (attacks) in each stage.

To make the Agent Manager and the Task Agents in the service to perform the mechanism, the service has to be executed in protection mode and the Task Agent should be written in appropriate format. Besides, a corresponding Checker should be provided.

To ease the transformation of normal Mobile Grid Service to Mobile Grid Service with agent protection, the MGS API provides a convert tool to generate appropriate "protected" Task Agent, Checker and related files from the user-defined Task Agent's Java file.

Services in the previous experiment involve random integer summations. They are not suitable for this experiment as the generated Checker can never re-execute with consistent results. Consequently, a new service called Service Y was implemented. When Service Y was requested, a new Task Agent would be created (without Monitor Agent). At home host (Host A), the Task Agent would carry out 10000 times the prepared task. The prepared task is 400000 times of summations. After that, the Task Agent would migrate to Host B and then Host C to carry out 10000 times the prepared tasks respectively. Finally, it would move to Host D and send final notification to Agent Manager.

In the experiment, Service Y (with normal Task Agent) was setup and invoked on Host A in secure mode and protection mode respectively. Their execution times were measured.

By using the convert tool in the MGS API, "protected versions" of the Task Agent and corresponding Checker were generated. They were used to setup the "protected version" of Service Y (with protected Task Agent) for carrying out the experiment. In fact, the tracing process would finish some times after the Task Agent finished its task. Thus, the times between the Task Agent initialization and the finish of whole tracing process was also measured.

For obtaining more information, the executions of Service Y (with normal and protected Task agent) were repeated by replacing the prepared task from 400000 times to 800000 times of summations. For each service and setting above, 10 trials were performed. The average results of the experiments are shown in Table 8-3.

8.4.2 Experiment results

In Table 8-3, the execution times of first invocation of all services are extracted from others and handled independently. The first invocation of the service always has an unusual long execution time than the invocation afterward because the JADE platform will optimize agent migration to known containers. For services using agent protection, the first invocation introduces a much larger variation. This is due to the creation of the checkers in each host. The time reserved for the checker to setup and move to the target location contributes mainly to the difference. After the first invocation, the checkers will not be killed and can be used by the invocation later. Thus, the execution time of subsequent invocation can be improved. To reduce the variation, the data of first invocation will not be considered in the following analysis.

Table 8-3: EXECUTION TIMES (IN MILLISECONDS) OF Service Y UNDER DIFFERENT SETTINGS

	Normal agent in Secure mode	Normal agent in Protection mode	Protected agent in Protection mode (task end)	Protected agent in Protection mode (tracing end)
400000 summations (1st invocation)	6101.0	6213.4	17997.0	18280.4
800000 summations (1st invocation)	8843.4	8876.8	19868.6	20095.2
400000 summations (exclude 1st invocation)	3987.9	3999.2	5111.8	5954.9
800000 summations (exclude 1st invocation)	6637.2	6674.1	7839.0	9477.2

From the data in Table 8-3, we can see that execution times for services with normal Task Agent in Secure mode and Protection mode are similar. That means Protection mode does not introduce significant overheads to the execution of normal Task Agent.

Figure 8-5 is plotted for the execution times of Service Y in protection mode under different settings. We can see that the agent execution requires longer time if protected agent is used instead of the normal one. This is due to the additional works such as trace creation and trace transmission completed by the protected agent. If we need to wait for the whole tracing process to finish, additional time is required. Although the tracing processes are mainly performed in parallel with the service execution, the re-execution of the last stage can be started only after the original service execution is completed. Therefore, additional time is required to wait for the tracing process to finish.

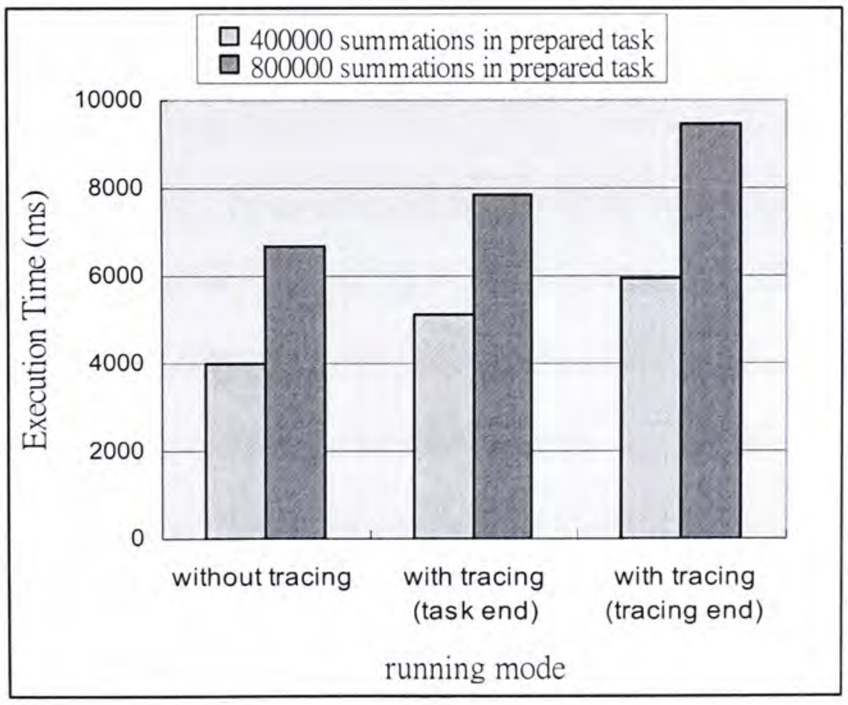


Figure 8-5: The execution time of Service Y in protection mode under different settings

Protected Task Agent with 400000 summations as prepared work requires

1112.6 milliseconds (27.82%) more than normal agent to get the final result and needs to wait an extra 843.1 milliseconds for the tracing to finish; while Protected Task Agent with 800000 summations as prepared work requires 1164.9 milliseconds (17.45%) more than normal agent and needs to wait an additional 1638.2 milliseconds for the tracing.

From the execution time difference, we can estimate the agent protection overhead per stage for the execution of protected Task Agent. Since the measurements cover three completed stages, the overhead is equal to [(1112.6 + 1164.9) / 2] / 3 = 379.6 milliseconds. On the other hand, the time for waiting for the whole tracing to finish cannot be estimated because it depends on the execution length of the last stage in Task Agent execution.

8.4.3 Discussions

In the MGS framework, execution tracing techniques are employed to achieve agent protection against malicious hosts due to its high feasibility, large scalability and relatively high accuracy. This approach makes use of re-execution to check for any attacks on the service execution. In our implementation, most of the re-execution is done in random hosts in parallel with the original Task Agent execution. This design not only allows the tracing process to be finished earlier (no need to wait double execution time), but also prevents the re-execution from delaying the original executions.

On the other hand, service developers can prepare simplified Checker for their own applications. For example, only re-execute the codes affecting an important variable and check for the final value of it. This can further speed up the tracing process.

From the experiment, we can see the overheads on the service execution when the agent protection support is used. The overheads are mainly contributed by the additional works of the Task Agent such as trace creation and stage commitment at each stage end (which will be ignited by migration). However, the number of migration should be small under normal situation and the service performance should not be degraded by the agent protection overheads.

Chapter 9

Conclusion and Future Works

In this thesis, the Mobile Grid Service Framework which supports Mobile Grid Services in a secure manner is presented. According to current grid technologies, standard Grid Services lack the runtime migration ability and have to stay at the home host even when idle resources are available throughout the grid. Static Grid Services can be enhanced by adding mobility and becoming Mobile Grid Services. The aims of this research are developing a middleware framework for grids to support secure Mobile Grid Services and providing facilities for easy and flexible service development.

The Mobile Grid Service Framework constructed by combining Java Agent Development Framework (JADE) [14] and Globus Toolkit [8] is introduced. Mobile Grid Services are realized as a special type of Grid Services which distribute the actual working task to mobile agents. This design makes Mobile Grid Services conform to the Globus Grid architecture while service mobility is achieved by the mobile ability of mobile agents in the services.

To support easy and flexible service development in the proposed framework, the MGS API consisting of AgentManager class, Configurable Monitor Agent, Task

Agent Templates and Resource Information Service is implemented. It provides libraries, templates and tools for service developers to develop their Mobile Grid Service.

To supplement the proposed framework, security support is added to ensure that Mobile Grid Service can execute securely. Taking the advantages of conforming to the Globus Grid architecture, globus-side security of the service can be fully reused. For agent-side security, security measures for mobile agents are deployed into the proposed framework. The details of these measures including authentication, authorization, message integrity and confidentiality, and agent permission mechanisms are presented.

To protect the agents in Mobile Grid Services from malicious hosts, an agent protection mechanism which is achieved by deploying "Execution Tracing with Randomly-Selected Hosts" into our framework is introduced. Re-execution is used to detect any malicious actions caused by hostile hosts.

Experiments have been conducted for evaluating the performance of Mobile Grid Services. Their results and analysis are discussed in this thesis and they show that the overload problem can be relieved by using Mobile Grid Services. The estimation of the general overheads, migration overheads, message overheads and agent protection overheads are also presented.

In the future, the Resource Information Service and the Configurable Monitor Agent in the MGS API will be improved to provide better load balancing performance. Extra resource metrics will be studied and added to the resource information exchange system in the Mobile Grid Service Framework such that the Monitor Agent can make appropriate migration decisions as soon as possible.

Appendix A

Administrator Guide for MGS API

In this appendix, a full installation of the MGS API and the setup of the MGS platform will be shown.

A.1 Installation of MGS API

A.1.1 Installation of pre-requisites

Install Java JDK

Download Java JDK5.0 from java homepage (http://java.sun.com) install them in all machines in the grid.

Install and configure Globus toolkit 4.0

Download GT4 from globus homepage (http://www-unix.globus.org/) and install them in all machines in the grid. Make sure that RFT service is configured well (i.e. no warning message during container starts).

Reference:

http://www-unix.globus.org/toolkit/docs/4.0/admin/docbook/quickstart.html

Install and configure Ganglia

Download and install the Ganglia monitor core v2.5.6 (newer version may not be compatible with GT4) from the following link:

http://prdownloads.sourceforge.net/ganglia/ganglia-monitor-core-gmond-2.5.6-1.i386.rpm?download

Then, run the ganglia tools using the command "gmond". A XML file containing the host's resource data will be generated.

(Can be tested by the command "telnet localhost 8649")

Reference: http://ganglia.sourceforge.net/docs/ganglia.html

To make GLOBUS receiving resource information from the ganglia, we should set the <defaultProvider> option in the file

"$GLOBUS_LOCATION/etc/globus_wsrf_mds_usefulrp/gluerp.xml"

as

<defaultProvider>java org.globus.mds.usefulrp.glue.GangliaElementProducer
</defaultProvider>

(Make sure that GLOBUS and postgresSQL are installed and configured well)

Install JADE and JADE-S

In all the hosts, download JADE v3.4 and JADE-S from http://jade.tilab.com and install it. However, since we have modified some code of JADE for our API, the pure version of JADE and JADE-S may get error during Mobile Grid Service execution. We have provided the modified JAR file for JADE and JADE-S in the ZIP file of our API. They are located in the "jadeLib/" folder.

To install JADE and JADE-S, unzip the API_v1.0.zip file (file for MGS API) in all machines as user "globus" (recommend to unzip it in /home/globus/mgs folder). Then, copy all the JAR files (including commons-codec-1.3.jar, jade.jar, jadeTools.jar, http.jar, iiop.jar, bcprov-jdk15-133.jar and jadeSecurity.jar) in the "jadeLib/" folder to $GLOBUS_LOCATION/lib directory.

Then, add all 7 JAR files to the CLASSPATH environment variable (Be careful for the files' privilege). This can be done by:

Method 1:

Modify /root/.bash_profile (permanent) by adding the follow lines:

```
CLASSPATH=$CLASSPATH:$GLOBUS_LOCATION/lib/commons-codec-1.3.jar:
$GLOBUS_LOCATION/lib/jade.jar: $GLOBUS_LOCATION/lib/jadeTools.jar:
$GLOBUS_LOCATION/lib/http.jar: $GLOBUS_LOCATION/lib/iiop.jar:
$GLOBUS_LOCATION/lib/ bcprov-jdk15-133.jar:
$GLOBUS_LOCATION/lib/jadeSecurity.jar
export CLASSPATH
```

Then run

```
source /root/.bash_profile
```

Method 2:

Using the following commands (temporary):

```
export CLASSPATH=$CLASSPATH:$GLOBUS_LOCATION/lib/jade.jar
export CLASSPATH=$CLASSPATH:$GLOBUS_LOCATION/lib/jadeTools.jar
...etc
```

(Check the environment variable by "echo $CLASSPATH" command)

Reference: JADE administrator's guide (Section 2.3)

(http://jade.tilab.com/doc/administratorsguide.pdf)

A.1.2 Installation of MGS API library

To install the MGS API library, unzip the API_v1.0.zip file (file for MGS API) in all machines as user "globus" (recommend to unzip it in /home/globus/mgs folder). Then, copy the mgs.jar into $GLOBUS_LOCATION/lib directory. Then, add this file to the CLASSPATH environment variable (using method above).

A.2 Setup of MGS platform

To setup a platform for the Mobile Grid Services, a JADE platform must be established in the grid to provide agents in the services with an execution environment. After that, Globus containers deployed with the Resource Information Service and the Mobile Grid Services should be started. The detailed procedures of setting up a MGS platform will be shows as follows:

A.2.1 Setup of JADE platform

In a trusted host, start the JADE main container by command:

```
java jade.Boot –conf <JADE config file>
```

(The JADE configuration file must be set to run as main container)

In other hosts (which want to provide resources), start the JADE containers by the same command:

```
java jade.Boot –conf <JADE config file>
```

(The JADE configuration file must be set to run as normal container and connect to the main container)

The JADE configuration file can be prepared by using the MGS Configuration Helper (Details in section B.5).

A.2.2 Setup of Globus containers

In a trusted host, deploy the Resource Information Service into local Globus container by:

```
$GLOBUS_LOCATION/bin/globus-deploy-gar org_mgs_resource.gar
```

(The org_mgs_resource.gar is the GAR file for the Resource Information Service which can be found by extract API_v1.0.zip file.)

Then, start the Resource Information Service and the corresponding Globus container by:

```
$GLOBUS_LOCATION/bin/globus-start-container
```

In other hosts where Mobile Grid Services are deployed, start the Globus containers by:

```
$GLOBUS_LOCATION/bin/globus-start-container
```

(Remember to configure the JADE configuration files for the creation of service container in the Mobile Grid Services)

Appendix B

Developer Guide for MGS API

In this appendix, the method of implementing and configuring a Mobile Grid Service, the useful tools provided and the interface of the MGS API are presented.

B.1 Steps of developing a Mobile Grid Service

This section shows the steps of developing a Mobile Grid Service.

B.1.1 Design Mobile Grid Service

Before developing a Mobile Grid Service, several questions should be considered first:

- How many operations should be provided in the service?
- How many kinds of task should be available?
- What each task should do?

B.1.2 Define WSDL

The WSDL file is used to define the interface of the service. We need to define well all operation elements (methods allow to be invoked by users) as well as the related message and type elements in this file

The details of preparing the WSDL file are described in the following Globus document:

http://gdp.globus.org/gt4-tutorial/multiplehtml/apa.html

B.1.3 Implement the service

Mobile Grid Service is composed of three components (Agent Manager, Task Agent and Monitor Agent). Therefore, the implementation of a Mobile Grid Service can be divided into three parts. Each part is responsible for the implementation of one component. The details of them will be described in section B.2.

B.1.4 Configure deployment in WSDD

The Web Service Deployment Descriptor (WSDD) is used to configure the deployment details of a service. In the WSDD file, developer needs to specify the class for service implementation (Agent Manager component), the WSDL file and the security descriptor (for specifying the security configuration of the service), etc. The details of preparing the WSDD file are described in the following Globus document:

http://gdp.globus.org/gt4-tutorial/multiplehtml/ch03s03.html

B.1.5 Compile and deploy the service

Compile the service

Developer can compile the service by a shell script. The command is:

```
./globus-build-service.sh -d <service base directory> -s <service's WSDL file>
```

Assume that the WSDL file is "schema/mgs/example/MyApplication.wsdl", the WSDD file is placed in the "org/mgs/example" folder and the Java files for the service implementation are placed in the "org/mgs/example/impl/" folder. The service can be compiled by the following command:

```
./globus-build-service.sh -d org/mgs/example
                    -s schema/mgs/example/MyApplication.wsdl
```

A GAR file (containing all the files and information the web server needs to deploy the service) will be generated. The name of the output file depends on the "service base directory" argument. In our example, the GAR file "org_mgs_example.gar" is generated.

Deploy the service into Globus container

Deployment is done with a GT4 tool that unpacks the GAR file and copies the files within (WSDL, compiled stubs, compiled implementation, WSDD) into key locations in the GT4 directory tree. After deploying the service, the service will be setup when the Globus container starts up.

The deployment command is:

```
$GLOBUS_LOCATION/bin/globus-deploy-gar <service's GAR file>
```

In our example, we can deploy the service by:

```
$GLOBUS_LOCATION/bin/globus-deploy-gar org_mgs_example.gar
```

To undeploy the service, we can use the command:

```
$GLOBUS_LOCATION/bin/globus-undeploy-gar org_mgs_example
```

B.2 Mobile Grid Service Implementation

Mobile Grid Service is composed of three components (Agent Manager, Task Agent and Monitor Agent). The implementation of each component will be described as follows:

B.2.1 Implement Task Agent

This step is responsible for the implementation of Task Agent in the service. The main task of the Mobile Grid Service should be implemented in the Task Agent component. It can be done with the help of the Task Agent Templates (in mgs/template/ folder) in the MGS API. The implementation methods for Task Agent with and without agent protection are different. Therefore, they will be considered separately:

Task Agent without agent protection (can run in any mode)

The implementation of Task Agent must extend the *TaskAgent* class. Developer can use the TaskAgent Template or the SimpleTaskAgent Template to implement the Task Agent. The security libraries can be used only for secure mode.

<u>For using the TaskAgent Template</u>

Three files including *TaskBehaviour.java*, *MyTaskAgent.java* and Task*ServeIncomingMessagesBehaviour.java* are required to be considered. Developer should implement the Task Agent (following the guidelines inside the template file) as the steps below:

1. Implement service task in the file *TaskBehaviour.java*. Multiple *TaskBehaviours* can be implemented if multiple different tasks will be performed in the service.
2. Rewrite the *TaskServeIncomingMessagesBehaviour.java* in order to specify the ACL message handling for extra request from Agent Manager.
3. Create newly-defined *TaskBehaviour* objects in MyTaskAgent.java. Also, add user-defined instance variables and methods (if any) to the indicated location in this file.

<u>For using the SimpleTaskAgent Template</u>

Only the *MySimpleTaskAgent.java* file (in mgs/template/ folder) is required to be considered. Developer should implement the Task Agent (following the guidelines inside the template file) as the steps below:

1. Declare user-defined instance variables and methods (if any) in the indicated location.
2. Implement service task in the *normalExecution()* method.
3. Specify stopping criteria of the task in the *endChecking()* method.
4. Implement the user-defined message handling in the *msgHandler()* method.

Task Agent with agent protection (can run in protection mode only)

The implementation of protected Task Agent (with agent protection) must extend the *ProtectedTaskAgent* class (comparing to the *TaskAgent* class for agent without agent protection). Besides, a class for Checker component (extending the Checker class) must be prepared for doing the re-execution. There are two methods to achieve this:

The first method is using convert tool provided in MGS API. The Task Agent must be implemented by using the SimpleTaskAgent Template first. Then, the Java files for corresponding protected Task Agent as well as Checker can be generated by using the convert tool provided in MGS API. (Details in section B.3)

The second method is directly implementing Task Agent and Checker by extending the *ProtectedTaskAgent* class and the *Checker* class respectively. All the abstract methods in the *ProtectedTaskAgent* class (the *normalExecution()*, *msgHandler()*, *endChecking()*, *getFinalState()*, *getInitState()* and *updateInitState()* methods) and the *Checker* class (the *normalExecution()*, *msgHandler()*, *endChecking()*, *getFinalState()*, *restoreState()* and *stateEquals()* methods) must be implemented. For the details of these abstract methods, please read the section B.6.3 and B.6.4.

Common notices

- During the implementation process, developer can use the library in the *TaskAgent* class to develop the service task. Security library is also provided, but it can be used only when the Task Agent runs in secure mode. The details of the provided library are described in section B.6.3.
- Developer should remember to change the file name and the class name when using the templates so that no duplicate classes are implemented.
- Message protection in the Task Agent can be configured by changing the settings in the *TaskSetting* object which is a parameter of the *TaskAgent* class (Details in section B.4.1).
- After the implementation, the codes for Task Agent implementation should be compiled and then packed in a JAR file by using the following command:

  ```
  jar cvf <name of output file> <folder containing the file>
  ```

  The output JAR file should be placed to $GLOBUS_LOCATION/lib/ folder.

B.2.2 Implement Monitor Agent (optional)

A default Monitor Agent (in mgs/monitor/ folder) is provided in the MGS API already. Migration decision's policy in the Monitor Agent can be configured by changing the settings in the *MonitorSetting* object which is a parameter of the *MonitorAgent* class (Details in section B.4.2). The default policy of migration decision is:

If the CPU value of current host is less than 9000 (i.e. 90 per cent of best expected clock rate in the grid), the host with best CPU resource will be chosen as migration target (where memory and harddisk resources are ignored). Otherwise, no migration will be performed.

If the default Monitor Agent cannot fulfill your requirement in the migration decision, you can also develop your own Monitor Agent by extending the provided one. The required work is just overriding the *decide()* method in the *MonitorAgent* class.

B.2.3 Implement Agent Manager

In the implementation of the Agent Manager component, an *AgentManger* object must be created. By using the methods provided in *AgentManager* class, we can create the Task Agent (with Monitor Agent or not), send command (in form of ACL message) to Task agents and carry out other actions.

Each Mobile Grid Service should provide some service operations (defined in WSDL) which allow service requestors to invoke. The implementation of those operations is done in this step. The provided methods in the *AgentManager* class should be use to achieve it. For example,

- Task Agent can be created to by using the *createAgent()* method (or methods in same series).
- Command can be sent to the created Task Agent to perform specific task by the *sendACL()* method (or methods in same series).
- The execution result of the Task Agent can be obtained by the *getFinalResult()* method.

The details of the provided methods in the *AgentManager* class are described in section B.6.1. Fig B-1 shows an example of the implementation of the Agent Manager component.

```
public class MyApplication{
      AgentManager manager;
      public MyApplication(){
            ...
            manager = new AgentManager("serviceSetting.xml");
      }
      public method1(){       //start Task Agent
            manager.createAgent("task1", "myPath.MyTaskAgent", null);
      }
      public method2(){       //ask Task Agent to do something
            ACLMessage msg = manager.createACLMessage("task1",
                  ACLMessage.REQUEST, "stop_counting");
            String reply = manager.sendACL(msg);
            ...      // handling of the reply
      }
      ...
}
```

Figure B-1: An example Mobile Grid Service (Agent Manager component)

More information of how to implement a Grid Service can be found in this link:

http://gdp.globus.org/gt4-tutorial/multiplehtml/ch03s02.html

B.3 Convert tool

It is a Java program provided in the MGS API (in tools/ folder) for preparing a protected version of Task Agent (with agent protection) from a Task Agent (without agent protection). This program will read a Java file of Task Agent (specified by user) and use the details (e.g. instance variables, methods and task implementation) to generate the essential files required for protected Task Agent creation automatically. The input Task Agent must be developed by using the SimpleTaskAgent Template because the program recognizes different parts of code (e.g. user-defined variables, user-defined methods and initialization) according to the marks in the template.

After running the program, three files will be created as follow:

- A protected Task Agent which is the protected version of the input Task Agent (extending *ProtectedTaskAgent* class).
- A Checker corresponding to the input Task Agent (extending *Checker* class). The generated Checker will do the re-execution by repeating the execution in Task Agent (i.e. no simplification for the re-execution).
- An *AgentState* class which is used to store all instance variables in the protected Task Agent for serialization.

To run the convert tool program:

```
java ConvertTool <input file> <output class name>
```

For example, if the program is run by command "java ConvertTool MyAgent.java NewAgent", three files named as *NewAgent.java*, *NewAgentChecker.java* and *NewAgentState.java* respectively will be generated.

B.4 Service configuration

B.4.1 TaskSetting object

A *TaskSetting* object is required to pass as argument when the *createAgent()* method (or method in same series) of *AgentManager* class is called. The *msgNeedEncrypt* and *msgNeedSign* variables in the object are used to decide the ACL message protection level for the Task Agent. The *sendMessage()* method in the Task Agent will send ACL message according to the *msgNeedEncrypt* (perform message encryption) and *msgNeedSign* (perform message signing) flags. The configuration method is invoking setMsgNeedSign(true) or/and setMsgNeedEncrypt(true) methods of the *TaskSetting* object before passing the object to the *createAgent()* method.

B.4.2 MonitorSetting object

A *MonitorSetting* object is required to pass as argument when the *createAgentwithMonitor()* method of AgentManager class is called. It is used to configure the execution mode, migration decision and message protection of the Monitor Agent created.

By setting the *min_CPU*, *min_RAM* and *min_HD* variables, the minimum requirements on the current host's resources (such that no migration is required to perform) can be configured. Besides, the weight_CPU, weight_RAM and weight_HD variables are used to determine the importance weights of each kind of resources (CPU, memory, and harddisk) in the migration decision. The calculation of the score of each host is done by the follow formula:

Score = (weight_CPU * CPU) + (weight_RAM * RAM) + (weight_HD * HD)

The host with the highest score will be chosen as the new execution host. Therefore, the migration decision policy can be adjusted by configuring the importance weights.

The *msgNeedEncrypt* and *msgNeedSign* variables in the object are used to decide the ACL message protection level for the Monitor Agent. The Monitor Agent will send resource data (in form of ACL message) according to the *msgNeedEncrypt* (perform message encryption) and *msgNeedSign* (perform message signing) flags.

B.4.3 MGS Configuration file

The configuration of each Mobile Grid Service is relying on the MGS configuration file. In each service implementation, the file name of the corresponding MGS configuration file should be passed as argument during the creation of *AgentManager* object. Fig B-2 shows an example of MGS configuration file.

```
<service>
        <Application_name>CounterService</Application_name>
        <SecureMode>true</SecureMode>
        <ProtectionMode>true</ProtectionMode>
        <JadeConfigFile>secure_service.conf</JadeConfigFile>
</service>
```

Figure B-2: Sample MGS configuration file

The configuration options in the MGS configuration file are described as follows:

The description of the MGS configuration file is as follows:

- Application_name

  It is used to specify the name of the service.

- SecureMode

It is used to specify whether the service run in secure mode. The security mechanisms (authentication, authorization, agent permission, message integrity and confidentiality, and agent protection) can be utilized only when the service run in secure mode.

- ProtectionMode

It is used to specify whether the service run in protection mode. The agent protection support can be utilized only when the service run in protection mode.

- JadeConfigFile

It is used to specify the name of the configuration file used for service container creation.

B.4.4 Configuration for Resource Information Service

The Resource Information Service can be configured by a XML file named resourceSetting.xml. Fig. B-3 shows a sample configuration file for the Resource Information Service.

```
<resource>
        <Max_CPU>3200</Max_CPU>
        <Max_RAM>2048</Max_RAM>
        <Max_HD>1024</Max_HD>
        <SendingPercent>25</SendingPercent>
        <SecureMode>false</SecureMode>
        <NeedSign>false</NeedSign>
        <NeedEncrypt>false</NeedEncrypt>
        <JadeConfigFile>resource.conf</JadeConfigFile>
</resource>
```

Figure B-3: Sample configuration file for the Resource Information Service

The configuration options in the Resource Information Service's configuration file are described as follows:

- Max_CPU (Unit: Megahertz)

  It is used to specify the best possible/expected CPU ClockSpeed in the whole grid. The value "3200" means that the best possible/expected CPU is 3.2 Gigahertz.

- Max_RAM (Unit: Megabyte)

  It is used to specify the best possible/expected RAM available in the whole grid. The value "2048" means that the best possible/expected RAM available is 2 Gigabyte.

- Max_HD (Unit: Megabyte)

  It is used to specify the best possible/expected Harddisk space available in the whole grid. The value "1024" means that the best possible/expected Harddisk space available is 1 Gigabyte.

- SendingPercent

  It is used to specify the chance (in percentage) that a host is chosen as target audience (in each round of data distribution). The value "25" means that the service will send resource data to about one-fourth of available hosts randomly in each round of resource information distribution (every 60 seconds).

- NeedSign

  It is used to specify whether the resource messages sent by the service need to be signed.

- NeedEncrypt

  It is used to specify whether the resource messages sent by the service need to be encrypted.

- JadeConfigFile

It is used to specify the name of the configuration file used for service container creation.

B.4.5 Globus-side security configuration of the service

The Globus-side security measures like authentication, authorization and message protection can be configured in the security descriptor. The details of the security descriptor can be found in the following link:

http://www-unix.globus.org/toolkit/docs/4.0/security/authzframe/security_descriptor.html

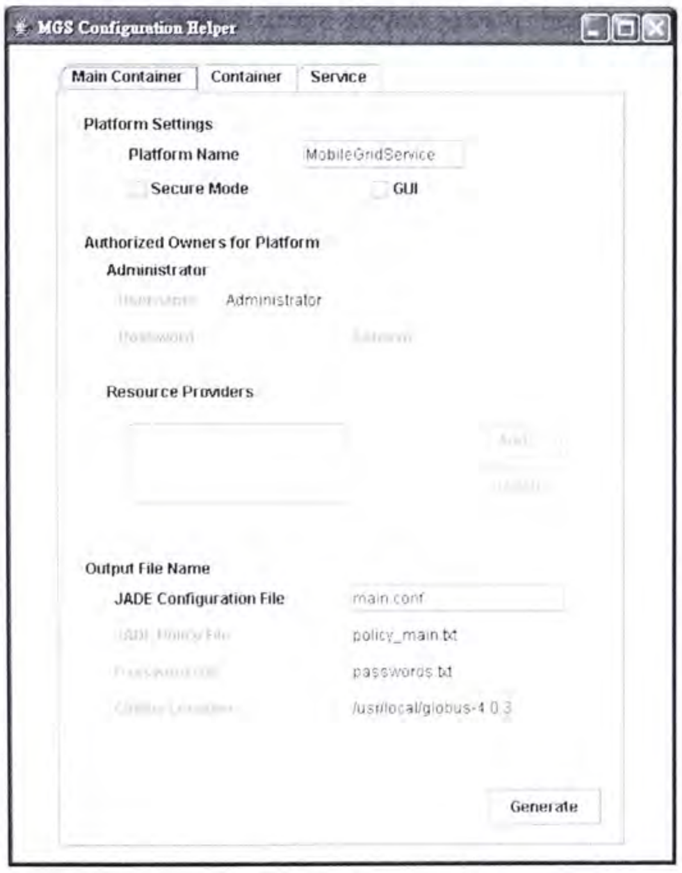
B.5 MGS Configuration Helper

It is a Java program for helping service and resource providers to handle the configuration works of the JADE platform. The JADE platform in the Mobile Grid Service Framework is formed by a main container with some containers (created by resource providers for agents to migrate on) connecting to it. During each service's initialization, a new service container will be created and connected to the JADE platform. Configuration of each container needs to handle a set of configuration files. We can imagine that configurations of Mobile Grid Services and the grid supporting them involve many configuration files. Consequently, the MGS Configuration Helper is provided to ease the configuration tasks. Service or resource providers can handle the JADE-side configuration works in a graphical user interface of the program instead of dealing with the configuration files separately.

The MGS Configuration Helper program is composed of three panels: "Main

Container", "Container" and "Service". They are responsible to help three different types of users respectively. The details of these three tagged panels are as follow:

B.5.1 "Main Container" Panel



Platform Settings

Platform Name: MobileGridService

Secure Mode

GUI

Authorized Owners for Platform

Administrator

Username: Administrator

Password

Resource Providers

Output File Name

JADE Configuration File: main.conf

JADE Policy File: policy_main.txt

Password File: passwords.txt

Globus Location: /usr/local/globus-4.0.3

Figure B-4: Graphical User Interface of MGS Configuration Helper (Main Container Panel)

Fig. B-4 shows the appearance of the "Main Container" panel. The administrator of the grid supporting Mobile Grid Services (i.e. administrator of the JADE platform) can use this panel to generate all configuration files for the JADE main container. The configuration works can be done by:

In the "Platform Settings" division

- Specify the name of the JADE platform in the "Platform Name" field.
- Tick the "GUI" check box if you want a graphical user interface to be launched on the main container (for controlling of the life-cycle of the JADE platform and all the registered agents).
- Tick the "Secure Mode" check box if you want the main container to run in secure mode

In the "Authorized Owners for Platform" division (only enabled when secure mode is chosen)

- Set the username and password of the administrator in the "Administrator" section.
- Add more valid owners (resource or service providers) to the platform by pressing the "Add..." button and then inputting the username and password of the new owners in the message dialogs coming out.

In the "Output File Name" division

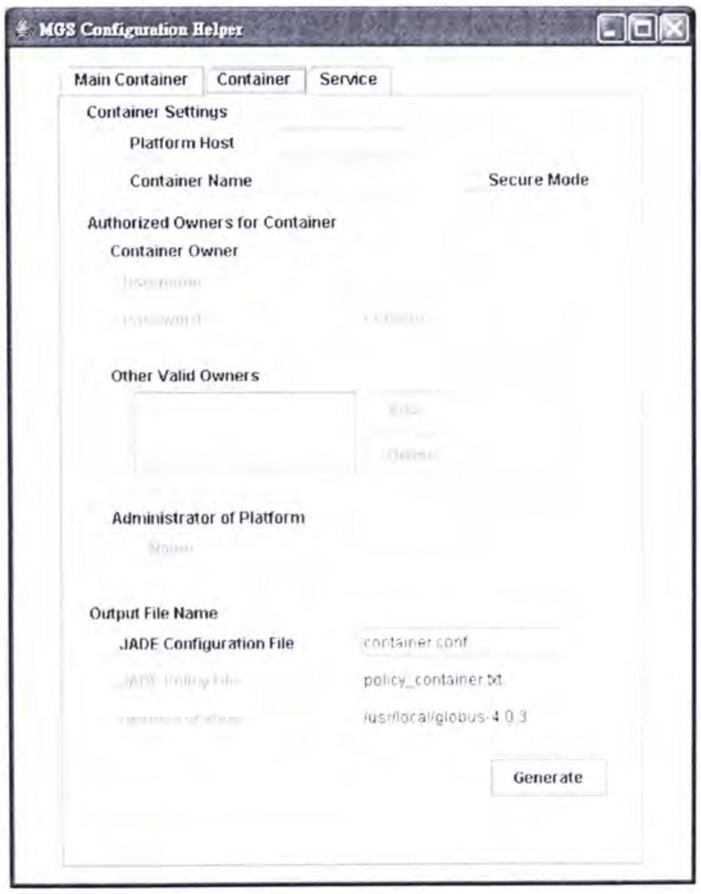
- specify the names of the output configuration files.

Finally, a full set of configuration files for the main container will be created by pressing the "Generate" button. For secure mode, these files include a JADE configuration file (for specifying container settings), a JADE policy file (for specifying the agent permission), a JADE password file (for storing passwords of each valid owner) and a JAAS configuration file (for specifying the login handling).

For non-secure mode, only the JADE configuration file will be created.

The agent permission in the JADE policy file is created according to the details of the valid owners. The generated file contains the basic permission such that all authorized owner are able to create agents on this container and communicate with Administrator. If more permission is needed to be set, the resource provider should directly modify the generated policy file.

B.5.2 "Container" Panel


MGS Configuration Helper

Main Container | Container | Service

Container Settings
- Platform Host
- Container Name
- Secure Mode

Authorized Owners for Container
- Container Owner
  - Username
  - Password
  - Confirm
- Other Valid Owners
  - Edit
  - Delete
- Administrator of Platform
  - Name

Output File Name
- JADE Configuration File: container.conf
- JADE Policy File: policy_container.txt
- [illegible]: /usr/local/globus-4.0.3

Generate

Figure B-5: Graphical User Interface of MGS Configuration Helper (Container Panel)

Fig. B-5 shows the "Container" panel of the program. The resource providers who want to contribute their resources in the grid (or the grid administrator) can use this panel to generate the configuration files required for their JADE containers. The configuration works can be done by:

In the "Container Settings" division

- Specify the host name of the administrator machine (where the main container is running on) in the "Platform Host" field.
- Specify the name of this container in the "Container Name" field (The host name of the machine hosting the container should be used).
- Tick the "Secure Mode" check box if the container runs in secure platform (i.e. main container is in secure mode).

In the "Authorized Owners for Container" division (only enabled when secure mode is chosen)

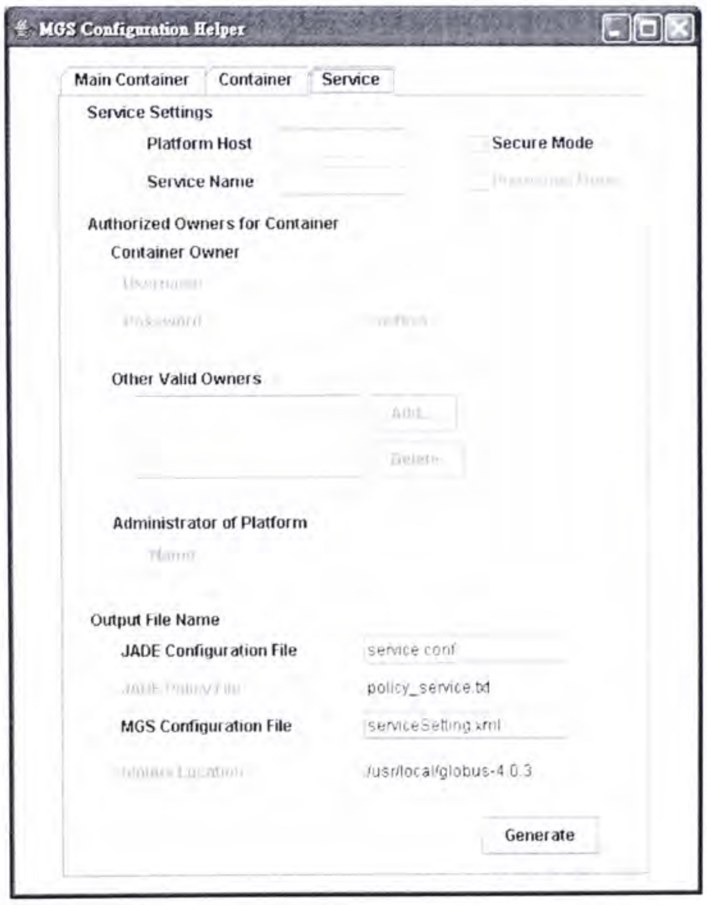
- Input the username and password of this container in the "Owner" section. The username and password should be already assigned to the resource providers such that they can successfully register to the JADE platform as a valid owner.
- Add the name of all valid owners (without password) into the "Other Valid Owners" section.
- Specify the administrator name of the JADE platform (i.e. identity of the grid administrator in the JADE platform).

In the "Output File Name" division

- Specify the names of the output configuration files

Finally, a JADE configuration file and a JADE policy file (for secure mode only) for the container will be created by pressing the "Generate" button. All the details in "Authorized Owners for Container" division will be used to prepare the basic permission policy for this container.

B.5.3 "Service" Panel


MGS Configuration Helper

Main Container | Container | Service

Service Settings

Platform Host

Service Name

Secure Mode

Authorized Owners for Container

Container Owner

Username

Password

Confirm

Other Valid Owners

Add...

Delete

Administrator of Platform

Name

Output File Name

JADE Configuration File: service.conf

JADE Policy File: policy_service.txt

MGS Configuration File: serviceSetting.xml

Globus Location: /usr/local/globus-4.0.3

Generate

Figure B-6: Graphical User Interface of MGS Configuration Helper (Service Panel)

Fig. B-6 shows the "Service" panel of the program. The service providers who want to provide their Mobile Grid Services in the Grid can use this panel to generate the configuration files required for their Mobile Grid Services. The configuration works can be done by:

In the "Container Settings" division

- Specify the host name of the administrator machine (where the main container is running on) in the "Platform Host" field.
- Specify the name of this service in the "Service Name" field. The container name will be automatically set by concatenating the host name (machine where the service resides on) to the service name, separated by the '_' character. For example, a service called Y deployed in a machine called A will create a container called "A_Y".
- Tick the "Secure Mode" check box if the container runs in secure platform (i.e. main container is in secure mode).
- Tick the "Protection Mode" check box (only enabled when secure mode is chosen) if the service needs to run in protection mode (i.e. the service supports agent protection).

In the "Authorized Owners for Container" division (only enabled when secure mode is chosen)

- Input the username and password of this container in the "Owner" section. The username and password should be already assigned to the resource providers such that they can successfully register to the JADE platform as a valid owner.
- Add the name of all valid owners (without password) into the "Other Valid Owners" section.
- Specify the administrator name of the JADE platform (i.e. identity of the grid administrator in the JADE platform).

In the "Output File Name" division

- Specify the names of the output configuration files

Finally, a JADE configuration file and a JADE policy file (for secure mode only) for the service container will be created by pressing the "Generate" button. The MGS configuration file for the service will be also generated. The "Output File Name" division is more important in the "Service" panel because multiple Mobile Grid Services may be deployed in a single machine. The file should be named properly such that each service has its own set of configuration files.

B.6 Interface details

B.6.1 Package mgs.manager

<u>Class AgentManager</u>

Constructor:

AgentManager(String configFileName)

Description:

Create its own JADE container and Agent Manager Agent. All implementation of Mobile Grid Services should create an AgentManager object and use this object to further create/communicate with other Task Agents.

Parameters:

configFileName – name/path of the MGS configuration file

Public methods:

1) **void createAgent(String agentName, String agentClass, Object[] agentArgs)**

Description:

Create a task agent. The reference of the newly created agent (AgentContainer) is stored in a hash table in the AgentManager object.

Parameters:

agentName - A platform-unique name for the newly created agent.

agentClass - The fully qualified name of the class that implements the agent.

agentArgs - An object array, containing initialization parameters to pass to the new agent.

Variants:

void createAgentWithMonitor(String agentName, String agentClass, Object[] agentArgs, MonitorSetting setting)

Description:

Create a task agent with a default monitor agent (configured with the input MonitorSetting). The name of MonitorAgent will be "Monitor-<agentName>".

void createAgentWithMonitor(String agentName, String agentClass, Object[] agentArgs, String monitorClass, Object[] monitorArgs)

Description:

Create a task agent with a specific monitor agent. Developers can use their self-defined MonitorAgent by this method.

void createProtectedAgent(String agentName, String agentClass, String checkerClass, Object[] agentArgs)

Description:

Create a protected task agent. The <checkerClass> specify the Checker for this Task Agent. (Used for protection mode only)

void createProtectedAgentWithMonitor(String agentName, String agentClass, String checkerClass, Object[] agentArgs, MonitorSetting setting)

Description:

Create a protected task agent with a default protected monitor agent. The <checkerClass> specify the Checker for this Task Agent. (Used for protection mode only)

2) **MGSResult getCurrentResult(String taskName)**

Description:

Get the current result of task agent from the resultTable

Parameters:

taskName - The name of target Task Agent.

3) **MGSResult getFinalResult(String taskName)**

Description:

Get the final result of task agent from the resultTable

Parameters:

taskName - The name of target Task Agent.

4) **AgentSituation getAgentSituation(String agentName)**

Description:

Get the current situation of specific agent. The result is returned as an AgentSituation object (includes agent state, position, etc).

5) **void moveAgentTo(String agentName, String containerName, String containerAddress)**

Description:

Force agent to move to a specific host (with monitor agent if any)

6) **ACLMessage createACLMessage(String agentName, int performative, String content)**

Description:

Create an ACL message by specify its receiver, performative and content.

Parameters:

agentName - The name of agent who should receive this ACL message

performative - The FIPA performative of the message (e.g. REQUEST, INFORM)

content – The content part of the message

7) **String sendACL(ACLMessage query)**

Description:

Send ACL message command or query to Task agent through Agent Manager Agent. The ACL message is neither signed nor encrypted.

Parameters:

query - The ACL message being sent.

8) **String sendSignedACL(ACLMessage query)**

Description:

Send ACL message command or query to Task agent through Agent Manager Agent. The ACL message is signed. (Used for secure mode only)

Parameters: **query** - The ACL message being sent.

9) **String sendEncryptedACL(ACLMessage query)**

Description:

Send ACL message command or query to Task agent through Agent Manager Agent. The ACL message is encrypted. (Used for secure mode only)

Parameters:

query - The ACL message being sent.

10) **String sendFullSecureACL(ACLMessage query)**

Description:

Send ACL message command or query to Task agent through Agent Manager Agent. The ACL message is both signed and encrypted. (Used for secure mode only)

Parameters:

query - The ACL message being sent.

11) **String checkingAtHome(String agentName)**

Description:

This method is designed for ensuring that tracing process of the specific Task Agent (specified by the argument <agentName>) can be completed After this method is called, a suitable Checker for that Task Agent will be created at the home host. All the unchecked stages will be assigned by the home Checker. Thus, all remaining re-execution can be done on the home host and the tracing can be finished. (Used for protection mode only)

Class AgentManagerAgent

Description:

Relay between Agent Manger and Task agents.

(Developers need not to care about this class)

Class MGSResult

Constructor:

MGSResult(String result, boolean finished, boolean checked, long startTime, long endTime)

Description:

Create a MGSResult object which will be stored in the Result Table.

Parameters:

result – The result string to be stored in the MGSResult object.

finished – Boolean value indicating whether the result is final result.

checked – Boolean value indicating whether the result is verified.

startTime – Long value for the start time of the Task Agent

endTime – Long value for the end time of the Task Agent

Public methods:

void setResult(String result)

Description:

Set the result field in the MGSResult object.

String getResult()

Description:

Return the result part in the MGSResult object.

boolean isFinished()

Description:

Return a boolean value indicating whether the result is final result.

boolean isChecked()

Description:

Return a boolean value indicating whether the result is verified.

long getStartTime()

Description:

Return a long value which is the start time of the Task Agent.

long getEndTime()

Description:

Return a long value which is the end time of the Task Agent.

Class AgentSituation

Constructor:

AgentSituation(String agentState, String agentLocation)

Description:

Create an object store the situation of an agent. Information include agent state and location

Public methods:

String getAgentState()

Description:

Return the agent state in the AgentSituation object.

String getAgentLocation()

Description:

Return the agent location in the AgentSituation object.

String toString()

Description:

Return the agent state in the AgentSituation object.

B.6.2 Package mgs.monitor

<u>Class MonitorAgent</u>

Description:

This class extends the TaskAgent class. It will wait to receive ACL message from Resource Information Service, update data, and then call decide() method to do the migration decision at appropriate time.

Constructor:

MonitorAgent(MonitorSetting setting, String taskAgentName)

Description:

Add MonitorServeIncomeMessagesBehaviour to the agent

Parameters:

setting – contains setting of the Monitor Agent (debug mode, GUI or not, other parameters to control the migration decision logic)

taskAgentName – The agent name of corresponding Task Agent.

Public methods:

String decide()

Description:

Method called to make the migration decision; return the target containerName or null if not decide to move. If developer requires a tailor-made logic, he can implement a new MonitorAgent by extending this class and override this method.

Class ProtectedMonitorAgent

Description:

This class extends the ProtectedTaskAgent class. It is the protected version of MonitorAgent. Except the agent protection support, other details are just the same as those of MonitorAgent class.

Class ProtectedMonitorAgentChecker

Description:

This class extends the Checker class. It is the Checker for the ProtectedMonitorAgent class.

(Developer need not to care about this class)

Class MonitorSetting

Description:

Create a MonitorSetting object with default value.

Constructor:

MonitorSetting ()

Table B-1: INSTANCE VARIABLES OF MonitorSetting CLASS

Variable	Type	Default	Description
min_CPU	int	9000 (90%)	Minimum CPU requirement on current host
min_RAM	int	0	Minimum memory requirement on current host
min_HD	int	0	Minimum harddisk requirement on current host
weight_CPU	double	1	Weight of CPU value on best host decision
weight_RAM	double	0	Weight of memory value on best host decision
weight_HD	double	0	Weight of harddisk value on best host decision
debugMode	boolean	false	Indicate whether the Monitor Agent run in debug mode
msgNeedSign	boolean	false	Indicate whether the message sent by Monitor Agent requires signature
msgNeedEncrypt	boolean	false	Indicate whether the message sent by Monitor Agent requires encryption
secureMode	boolean	false	Indicate whether the Monitor Agent run in secure mode (set by Agent Manager)

Public methods:

Get and Set methods for all variables in the table above.

(e.g. **getMin_CPU(), setMin_CPU(int min_CPU)**)

B.6.3 Package mgs.task

Class TaskAgent

Constructor:

TaskAgent(TaskSetting setting)

Description: Create a Task Agent.

Parameters:

setting – contains setting of the Task Agent (secure mode, message need sign/encrypt or not)

Public methods:

public void notifyResult(String value)

Description:

This method is used to send the current result to Agent Manager. The result is sent through DF service notification.

public void notifyFinalResult(String value)

Description:

This method is used to send the final result to Agent Manager. The result is sent through DF service notification.

public void notifyConfidentialResult (String value)

Description:

This method is used to send the confidential final result to Agent Manager. It will set the confidential result in the agent first and then notify Agent Manager through DF service notification. After receiving request for the result, the confidential result will be sent to Agent Manager by encrypted message. (Used for secure mode only)

public void notifyConfidentialFinalResult (String value)

Description:

This method is used to send the confidential final result to Agent Manager. It will set the confidential result in the agent first and then notify Agent Manager through DF service notification. After receiving request for the result, the confidential result will be sent to Agent Manager by encrypted message. (Used for secure mode only)

public void setConfidentialResult(String value)

Description:

This method set the confidential result which will be obtained by Agent Manager. (Used for secure mode only)

public String getConfidentialResult()

Description:

This method returns the confidential result which is ready to be obtained by Agent Manager. (Used for secure mode only)

public void sendMessage(ACLMessage msg)

Description:

This method is used to send ACL message according to the msgNeedSign and msgNeedEncrypt flags in the TaskSetting object.

public void sendClearMsg(ACLMessage msg)

Description:

This method is used to send plain text ACL message no matter what the msgNeedSign and msgNeedEncrypt flags are.

public void sendSignedMsg(ACLMessage msg)

Description:

This method is used to send signed ACL message no matter what the msgNeedSign and msgNeedEncrypt flags are. (Used for secure mode only)

public void sendEncryptedMsg(ACLMessage msg)

Description:

This method is used to send encrypted ACL message no matter what the msgNeedSign and msgNeedEncrypt flags are. (Used for secure mode only)

public void sendFullSecureMsg(ACLMessage msg)

Description:

This method is used to send signed and encrypted ACL message no matter what the msgNeedSign and msgNeedEncrypt flags are. (Used for secure mode only)

public void moveMGS(Location nextHost)

Description:

This method is used to move the Task Agent to target location.

public ACLMessage receiveMGS(MessageTemplate pattern)

Description:

This method is used to receive ACL message.

<u>Class SimpleTaskAgent</u>

Description:

This class extends the TaskAgent class. It is implemented for easing the development of Task Agent.

Constructor:

SimpleTaskAgent(TaskSetting setting)

Description:

Create a Task Agent.

Parameters:

setting – contains setting of the Task Agent (secure mode, message need sign/encrypt or not)

Abstract methods:

protected abstract void msgHandler()

Description:

This method will be invoked for each iteration. The handling of any incoming ACL messages should be implemented. Developers should add the management of the user-defined messages.

protected abstract void normalExecution()

Description:

This method states the program logic of the service tasks.

protected abstract boolean endChecking()

Description:

This method is used to specify the stopping criteria of the Task Agent execution. If the stopping criteria are met, it should return true.

<u>Class ProtectedTaskAgent</u>

Description:

This class extends the TaskAgent class. It is the protected version of Task Agent. Agent protection mechanism will be performed during its execution.

Constructor:

ProtectedTaskAgent(TaskSetting setting)

Description:

Create a Task Agent.

Parameters:

setting – contains setting of the Task Agent (secure mode, message need sign/encrypt or not)

Abstract methods:

protected abstract void msgHandler()

Description:

This method will be invoked for each iteration. The handling of any incoming ACL messages should be implemented. Developers should add the management of the user-defined messages.

protected abstract void normalExecution()

Description:

This method states the program logic of the service tasks.

protected abstract boolean endChecking()

Description:

This method is used to specify the stopping criteria of the Task Agent execution. If the stopping criteria are met, it should return true.

protected abstract byte[] getInitState()

Description:

This method is responsible for extracting the initial execution state from specific instance variables.

protected abstract byte[] getFinalState()

Description:

This method is responsible for extracting current execution state from the core instance variables.

protected abstract void updateInitState()

Description:

This method is responsible for updating the specific instance variables (used for extracting initial state) by the current values of the core variables. It will be invoked when a new stage starts.

Public methods:

public void setStageEnd()

Description: This method is used to specify a user-specified stage end. The current stage of Task Agent execution will end after invoking this method.

Class TaskSetting

Constructor: **TaskSetting ()**

Description: Create a TaskSetting object with default value.

Table B-2: INSTANCE VARIABLES OF TaskSetting CLASS

Variable	Type	Default	Description
msgNeedSign	boolean	false	Indicate whether the message sent by Task Agent requires signature
msgNeedEncrypt	boolean	false	Indicate whether the message sent by Task Agent requires encryption
secureMode	boolean	false	Indicate whether the Task Agent run in secure mode (set by Agent Manager only)
protectedMode	boolean	false	Indicate whether the Task Agent run in protection mode (set by Agent Manager only)
agentManager	AID	null	AID of the corresponding Agent Manager
appArguments	Object[]	null	Pass extra arguments for the application to the Task Agent

Public methods:

Get and Set methods for all variables in the table above.

(e.g. **getMsgNeedSign()**, **setMsgNeedSign(boolean msgNeedSign)**)

B.6.4 Package mgs.ftsFramework

Class Checker

Constructor:

Checker(TaskSetting setting)

Description:

Create a Checker.

Parameters:

setting – contains setting of the corresponding Task Agent

Abstract methods:

public abstract void normalExecution()

Description:

The program logic of the corresponding service tasks should be implemented in this method such that re-execution can be done during checking.

protected abstract void msgHandler()

Description:

This method will be invoked for each iteration. The handling of any incoming ACL messages should be implemented as same as those of the corresponding Task Agent. Unlike the one in the TaskAgent class, this method will obtain the incoming messages from the message log stored in the trace.

protected abstract boolean endChecking()

Description:

This method is used to determine whether the execution of the task is complete. It should be equal to the same method in the Task Agent.

protected abstract byte[] getFinalState()

Description:

This method is responsible for extracting current execution state from the core instance variables. It should be equal to the same method in the Task Agent.

protected abstract void restoreState(byte[] state)

Description:

The method will be used to restore the instance variables in the Checker to the condition recorded in the specified execution state.

public abstract boolean stateEquals(byte[] state1, byte[] state2)

Description:

This method is used to determine whether two execution states are equal. Since execution state for each Task Agent should be different, developer should provide method to check the equality of any two states. It should return true if the two states are equal.

Bibliography

[1] I. Foster, C. Kesselman, and S.Tuecke, The Anatomy of the Grid, Enabling Scalable Virtual Organizations, International Journal of Supercomputer Applications, 2001, pp.200-222.

[2] Ian Foster and C. Kesselman, editors, The Grid2: Blueprint for a New Computing Infrastructure, Morgan Kaufmann, 2004.

[3] Extensible Markup Language (XML).
URL http://www.w3.org/XML/

[4] R. Chinnici, M. Gudgin, J.-J. Moreau, S. Weerawarana, Editors, Web Services description language (WSDL) version 1.2, World Wide Web Consortium, 2003, http://www.w3.org/TR/2003/WD-wsdl12-20030303/

[5] Simple Object Access Protocol (SOAP)
URL http://www.w3.org/TR/SOAP

[6] I. Foster, C. Kesselman, J. Nick, and S. Tuecke, The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration, Technical report, Globus Project, 2002.

[7] Global Grid Forum.
URL http://www.gridforum.org

[8] Globus toolkit.

URL http://www.globus.org/tookit

[9] Globus Alliance.

URL http://www.globus.org

[10] Web Services Resource Framework (WSRF), http://www.globus.org/wsrf/specs/ws-wsrf.pdf

[11] Grid Security Infrastructure: A Standards Perspective

URL http://www.globus.org/toolkit/docs/4.0/security/GT4-GSI-Overview.pdf

[12] X.509 certificates.

URL http://www.ietf.org/rfc/rfc2459.txt

[13] The Foundation for Intelligent Physical Agents (FIPA).

URL http://www.fipa.org

[14] Java Agent Development framework (JADE).

URL http://jade.tilab.com/

[15] M. Fukuda, Y. Tanaka, N. Suzuki, L. F. Bic, and S. Kobayashi, A mobile-agent-based PC grid, in: Proceedings of Autonomic Computing Workshop Fifth Annual International Workshop on Active Middleware Services (AMS), 2003, pp. 142-150.

[16] N. Suri, P. T. Groth, J. M. Bradshaw. While You're Away: A System for Load-Balancing and Resource Sharing Based on Mobile Agents, in: Proceedings of the 1st International Symposium on Cluster Computing and the Grid (CCGRID), 2001, pp. 470-475.

[17] R. M. Barbosa and A. Goldman, MobiGrid - Framework for Mobile Agents on Computer Grid Environments, in: Proceedings of the first International Workshop on Mobility Aware Technologies and Applications (MATA), 2004, pp. 147-157.

[18] A. Goldchleger, F. Kon, A. Goldman, M. Finger, and G. C. Bezerra, InteGrade: Object-Oriented Grid Middleware Leveraging Idle Computing Power of Desktop Machines, in: Concurrency and Computation: Practice & Experience. Vol.16, pp. 449-459.

[19] Aglets.
URL http://aglets.sourceforge.net/

[20] N. Suri, J. M. Bradshaw, M. R. Breedy, P. T. Groth, G. A. Hill, R. Jeffers, and T. S. Mitrovich, An Overview of the NOMADS Mobile Agent System, in: Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP), 2000.

[21] N. Suri, J. M. Bradshaw, M. R. Breedy, K. M. Ford, P. T. Groth, G. A. Hill, and R. Saavedra. "State Capture and Resource Control for Java: The Design and Implementation of the Aroma Virtual Machine", in : Proceedings of the Java Virtual Machine Research and Technology Symposium, 2001.

[22] J. Hulaas, W. Binder, and G. D. M. Serugendo, Enhancing Java Grid Computing Security with Resource Control, in: Proceedings of the First International Conference Grid Services Engineering and Management (GSEM) 2004, pp. 30-47.

[23] OASIS. "UDDI Technical White Paper".
URL http://uddi.org

[24] W. Zhang, J. Zhang, D. Ma, B. L. Wang, Y. T. Chen, Key Technique Research on Grid Mobile Service, in: Proceedings of the Second International Conference on Information Technology for Application (ICITA), 2004, pp. 144-148.

[25] T. Ma, and S. Li, Secure Grid-based Mobile Agent Platform by Instance-Oriented Delegation, in: The Second International Workshop on Grid and Cooperative Computing (GCC), 2003, pp. 916-923.

[26] Globus Index Service.
URL http://www.globus.org/toolkit/docs/4.0/info/index/

[27] Ganglia monitoring system.
URL http://ganglia.sourceforge.net

[28] JAVA.
URL http://java.sun.com

[29] Java Authentication and Authorization Service (JAAS).
URL http://java.sun.com/products/jaas/

[30] G. Vigna, Cryptographic traces for mobile agents, in: Mobile Agents and Security, 1998, pp. 137–153

[31] T. Sander, C. F. Tschudin, Towards mobile cryptography, in: Proceedings of the IEEE Symposium on Research in Security and Privacy, IEEE Computer Society, Technical Committee on Security and Privacy, IEEE Computer Society Press, 1998.

[32] S. Funfrocken, Protecting mobile web-commerce agents with smartcards, Autonomous Agents and Multi-Agent Systems, Volumn 4, Issue 4, 2001, pp. 339–358.

[33] O. Esparza, M. Soriano, J. L. Muñoz, J. Forné, A protocol for detecting malicious hosts based on limiting the execution time of mobile agents, in: IEEE Symposium on Computers and Communications (ISCC), 2003, pp. 251–256.

[34] K.K. Leung, and K.W. Ng, Detection of Malicious Host Attacks by Tracing with Randomly Selected Hosts, Embedded and Ubiquitous Computing, Springer-Verlag, 2004, pp.839-848.

Publications

Accepted publications

Sze-Wing Wong and Kam-Wing Ng, A Middleware Framework for Secure Mobile Grid Services, in: International Workshop on Agent based Grid Computing at 6th IEEE International Symposium on Cluster Computing and the Grid (CCGrid), 2006.

- Corresponding to Chapter 3 and 4

Sze-Wing Wong and Kam-Wing Ng, MGS: An API for Developing Mobile Grid Services, in: OnTheMove Federated Conferences (OTM), 2006, LNCS 4276, pp.1361-1375, 2006

- Corresponding to Chapter 5

Sze-Wing Wong and Kam-Wing Ng, Security Support for Mobile Grid Service Framework, in: International Conference on Next Generation Web Services Practices (NWeSP), 2006.

Sze-Wing Wong, and Kam-Wing Ng, Security Support for Mobile Grid Services, in: International Journal of Web Services Practices (IJWSP), 2006.

- Corresponding to Chapter 6