General Architecture for Demand Migration in the GIPSY Demand-Driven Execution Engine

Emil Iordanov Vassev

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Computer Science at

Concordia University

Montreal, Quebec, Canada

June 2005

Your file Votre référence
ISBN: 0-494-10296-9
Our file Notre référence
ISBN: 0-494-10296-9

Abstract

General Architecture for Demand Migration in the GIPSY Demand-Driven Execution Engine

Emil Vassev,
Concordia University, 2005

Intensional programming is a form of demand-driven dataflow computation based on intensional logic. The last is a family of mathematical formal systems that allow expressions whose value depends on hidden context or indices. There is a wide range of software applications based on intensional logic from different computing areas including parallel programming, dataflow computation, temporal reasoning, scientific computation, real-time programming etc. The General Intensional Programming System (GIPSY) is a member of this family. The GIPSY exposes a demand-driven execution model with essence of distributiveness. The GIPSY programs are evaluated on different machines hosting the GIPSY execution nodes. The last generate and execute functional demands.

In this thesis, we discuss the solution to the problem of migrating demands among the GIPSY execution nodes. We emphasize the concept of a generic framework for migrating demands and apply this framework for creating a Demand Migration System (DMS). We describe the design and implementation of the Demand Migration System, and survey the test results. We also explore the distributed middleware technologies JINI, CORBA, DCOM and .Net Remoting, those being the implementation backbone for our DMS. Our thesis covers only the demand migration aspects and does not deal with load balancing and efficiency aspects of the GIPSY, as these are to be tackled by other subsystems of the GIPSY and future subsystems of the DMS.

This thesis represents the blending of two viewpoints: that of the practitioner, whose main goal is to build high-quality software, and that of the researcher, who strives to make a discovery or invent new aspects of computing.

to my beloved wife Slava

and to my magnificent kids Danko and Emili

Acknowledgements

I am very grateful to the many people who helped with this thesis and provided valuable support.

First, I would like to express my deep gratitude to my esteemed supervisor, Dr. Joey Paquet for his invaluable guidance and thorough support during my graduate study. I am indebted to Dr. Paquet not only for being my supervisor, but also my mentor and open-minded friend. He has shown me that research is well beyond writing thesis and publications, but also collaboration with people.

Special thanks to all the members of the GIPSY team for their support and comments during the years of my graduate study.

In addition, I would like to thank the University of Concordia in Montreal and especially the Faculty of Computer Science for opening a door.

Finally, this thesis would not be possible without the encouragement and never ending support of my beloved wife Slava and my dearest parents.

And of course my kids Danko and Emili for being patient with their constantly busy Daddy.

Table of Contents

List of Figures

List of Tables

List of Listings

Chapter 1: Introduction

Everything should be made as simple as possible, but not simpler.
Albert Einstein

This chapter introduces the background of the Demand Migration System that we designed and developed as a part of the GIPSY project, highlights the contribution of the thesis and outlines its structure.

1.1. Context

The General Intensional Programming System (GIPSY) is a multi-language programming environment and demand-driven execution environment [1]. The GIPSY is aimed at the long-term investigation into the possibilities of intensional programming, a declarative and functional family of programming languages [5, 35, 36, 37, 38]. The GIPSY is an ambitious project directed by Dr. Joey Paquet and Dr. Peter Grogono in the Computer Science & Software Engineering Department at Concordia University in Montreal.

This thesis will focus on the execution aspect of the GIPSY, more specifically on the architecture used for the distributed migration and evaluation of the demands generated by the system at run-time. From this perspective, the GIPSY evaluation engine, or GEE (General Eduction Engine) is a Demand-Driven Execution System that is based on Demand Generators (DGs) that determine the control process by generating functional demands and workers that execute them [1]. The DGs and workers are the GIPSY execution nodes, and these nodes are potentially remote ones, in which case the demands are migrated via the network from the generators to the workers and responses flowing in reverse order. It is important to note that all the functional demands generated by the GIPSY are atomic and independent in the sense that they do not share a common state.

In the course of this thesis, we investigated the demand-migration problem, and formalized a GIPSY Demand Migration Framework (DMF). The DMF establishes a scheme for connecting the GIPSY execution nodes together using a generic architecture and enabling the use of different middleware technologies. Further, by applying this framework we designed and developed a Demand Migration System.

1.2. Thesis Statement

The GIPSY brings to its clients the benefits of distributed computing, where the generators and workers are located on different machines, and form together a demand-driven system that relies on the calculation power of all the CPUs and memory in the GIPSY network. There are several important issues related to the demand-driven execution model, but in this thesis we put accent on its control flow, which is driven by dependencies between demands, as well as special demands, "functional demands" that are at the leaves of the demand tree [1, 11, 12]. The problem statement of this thesis is how to connect the GIPSY generators with workers, or how to migrate the demands from one execution node to the other in a heterogeneous and distributed execution environment. This thesis work focuses on building a generic Demand Migration System (DMS) that addresses these issues. Our primary goal is a generic framework extensible enough to adopt arbitrary distributed middleware technologies and flexible enough to overcome the obstacles raised by their co-existence. In addition, we aim at a DMS following the principles of high scalability, reliability and security, those being from great importance to the computing communication systems.

1.3. Contributions

This thesis aims at the construction of an effective Demand Migration System (DMS) for the GIPSY demand-driven execution engine. The thesis designs and implements a demand migration model by applying distributed middleware technologies. This model brings the GIPSY to a high level of distributiveness and interoperability of the operational nodes. Contributions of the thesis range from the conceptual to design and implementation points of view.

The thesis conceives the conceptual aspects of a generic Demand Migration Framework (DMF) that determines the guidelines for the creation of a DMS. Such a DMS must be able to work in a heterogeneous environment and able to migrate real objects among the GIPSY operational nodes. The DMF exposes a layered structure, where the different layers are designed and implemented as loosely coupled components interacting through asynchronous messages. The DMS' components are volunteers, able to plug in and plug out the DMS at any time. Hence, the DMS

exposes an extensible architecture that is easily manageable, and adaptable to future changes.

The thesis gives details about the design and implementation of the DMS, by exploring four different design views of the DMS and analyzing the implementation aspects [18]. The set of design views includes a scenario view, logical view, process view and deployment view, where each view depicts different aspects of our design solution. Three example problems are then programmed and their tests results are surveyed in order to demonstrate that the system fulfils the requirements for heterogeneity, parallelism, scalability, reliability and accuracy.

In this thesis, a new computing communication model is introduced. This model is based on the distributed middleware paradigm and on the co-existence of technologies implementing this paradigm. Such technologies are JINI, CORBA, DCOM and .Net Remoting. The communication model is similar to the persistent asynchronous message passing communication model [45], where messages are delivered on demand. The biggest advantage of our model is the ability to transport not only data but also real objects, i.e. the system transports data and behavior. The system is built for the purpose of the GIPSY but is practically independent. The DMS demonstrates high interoperability, and with this level of interoperability, we can easily conceive an independent system.

The main disadvantage of our solution is the high latency, which is inherited from the applied distributed middleware. The high latency of message transmission is the major challenge in worldwide distributed computing. We minimize the latency by implementing synchronous messaging or peer-to-peer communication mechanism, but these will reduce the high scalability of our solution.

1.4. Structure of the Dissertation

This thesis is organized in three parts. Part I (Chapters 1 and 2) introduces the reader to the problem domain and gives a conceptual view of our solution. Part II (Chapters 3 and 4) walks through the major aspects of the system design and implementation. Part III (Chapters 5 to 8) focuses on evaluation and conclusion.

Chapter 2. This chapter introduces the context of the problem domain, and builds a conceptual view of our solution. Here, we explore the Demand Migration Framework

and the application of the framework resulting in a Demand Migration System. With this chapter, we motivate the reader to go over the design and implementation aspects of the solution described by the following chapters.

Chapter 3. In this chapter, we present our object oriented DMS design. The chapter describes four different design views of our DMS. The first view, called scenario view, captures the use of the system in terms of various use-cases. The second view, called logical view, is about the architectural and detailed design. The third view, called process view, captures the run-time aspects of the DMS. Moreover, the fourth view, called deployment view, describes the physical distribution of the DMS' components.

Chapter 4. In this chapter, we present a comprehensive overview of the DMS implementation. The chapter describes the most important implementation aspects of our solution, and makes comments on some specific parts of the code. In addition, here we present the distributed middleware technologies from an implementation perspective.

Chapter 5. In this chapter, we present the experimental results obtained in the course of this research. The chapter describes selected test examples that address important issues like heterogeneity, parallelism, scalability, reliability and accuracy.

Chapter 6. This chapter presents related to our research work.

Chapter 7. This chapter concludes our thesis.

Chapter 8. This chapter presents our future work.

Chapter 2: Conceptual View

The hardest part of building a software system is deciding precisely what to build.
Fredrick Brooks

This chapter describes the conceptual view that we built as a direct consequence of the requirements analysis undertaken in the course of this thesis. The conceptual view of the Demand Migration Framework (DMF) identifies the high-level concepts of the framework architecture. These concepts necessitate high-level components of the scheme, established by the DMF, and the relationships among them. Therefore, the conceptual view directs attention at an appropriate decomposition of the system without delving into details. Moreover, it provides an architecture perspective of the system aimed to non-technical audiences.

The chapter starts with the requirements analysis, and goes over the DMF generic architecture and its application in the form of a Demand Migration System (DMS). Finally, the chapter concludes with a short description of those distributed middleware concerns that have influenced our DMF design solution.

2.1. Requirements

As a result of the demand-migration problem investigation, we elicited a set of requirements that must be fulfilled by the demand-migration scheme established by DMF. The following elements specify a set of vital requirements that the Demand Migration System (DMS) must fulfill.

Platform interoperability. GIPSY programs are evaluated on multiple platforms. Hence, the DMS serving the GIPSY nodes (run on separate machines) should be able to deal with the process-machine boundaries and with the diversification of the different platforms, i.e. DMS should be able to connect the GIPSY nodes run on Linux\Unix, Solaris, Windows and Mac-OS platforms, using different middleware technologies available on these different platforms.

At least once delivery semantics. The GIPSY nodes run independently from each other. Hence, the DMS should be able to connect these artifacts at any time and

assures *at least once-delivery semantics* [45], i.e. no demand or result could be delivered on a wrong address and must be delivered at least once.

Asynchronous communication. Since the GIPSY nodes are not synchronized – they run independently and their lifetime is not synchronized, the DMS must perform *asynchronous communication* where the nodes do not connect permanently and do not synchronize their data exchange.

No demands discrimination. Since the demands generated by generators are atomic with no dependency in the sense of data sharing and time, the DMS should not discriminate them in terms of importance. These efficiency-related considerations are to be tackled by other parts of the GIPSY and future subsystems of the DMS.

No workers discrimination. A worker must be able to serve any generator, i.e. pending for execution demands should not wait more than the time needed for their delivery to the first free worker. Hence, the DMS must present the workers as a common set to all the generators with no discrimination in terms of importance or capacity to respond to functional demands.

Secure communication. Since the GIPSY nodes are located on different machines, we need to use security mechanisms to authenticate the identity of the DMS objects. The DMS should integrate a security mechanism.

Fault-tolerant demand migration. When objects are distributed across process boundaries, the objects can fail independently. Similarly, in a distributed system the network can interrupt or the system can be partitioned into disconnected parts, and components can run independently assuming the others have failed. In our Demand Migration System (DMS), there must be concerns about the behavior of the overall system if some of the components are available and others not. The demand-driven execution model permits such kind of better fault-tolerance. The DMS should keep track of the demands, so that the failure of any node does not result in loosing demands, and that node failure would simply result in re-issuing of the demands.

Distributed technology independency. All the requirements stated above necessitate a system that adheres to the characteristics of distributed computing and asynchronous communication with a certain security and permanent storage mechanism. There exists a wide array of distributed execution platforms and middleware technologies. Meeting all these requirements necessitates a very general and flexible approach that is not bound to a specific technology, and that can enable the use of different technologies. One of the main principles of the GIPSY is the platform independency, i.e. the DMS must be flexible and structurally generic to work with most of these distributed computation technologies and implementation platforms.

Hot-plugging. The GIPSY model of computation is not only a distributed demand-driven one, but also one in which all the nodes are "volunteers" that register to a dispatcher node, and that are later assigned a role and grafted to the network. Any node, including the DMS' nodes, has to be designed to allow the "hot-plugging" of new nodes, i.e. to add new nodes as the execution is taking place.

Upgradeability. The DMS should be designed in a manner that will allow the GIPSY clients the power of using their own distributed computation technology, i.e. the DMS should not be bound to any distributed execution technology and be generic enough as to allow the use of other technologies not part of its original implementation.

2.2. Framework Conceptual Model

Given the requirements analysis of the last section, the conclusion about the DMF architecture is a generic scheme for migrating objects – demands and results, in a heterogeneous and distributed environment that being specified by the GIPSY nodes. Therefore, the DMF establishes the context for performing the migration activities, i.e. it is about the process migration [10].

2.2.1. Framework Architectural View

For the architectural model of our DMF, we propose a layered structure, which helps the functionality of the system to be implemented in several layers. Fig.2.1 represents

an abstract conceptual view of our DMF, which uses a layered architecture style. The largest circle depicts the GIPSY, and the double-lined inner circle depicts the DMF. The GIPSY is represented as a set of operational nodes - workers and DGs, those being the communication end points, and the DMF acting as a communication intermediate between them. The DMF consists of two principal functional layers called Demand Dispatcher (DD) and Migration Layer (ML). The DD (depicted by a bold-lined circle in the Fig.2.1) is an object storage mechanism able to dispatch the objects to their recipients. The ML (depicted as a dark grayed layer on top of DD) is the layer performing the object migration from the DD to the recipient GIPSY nodes – workers or DGs.

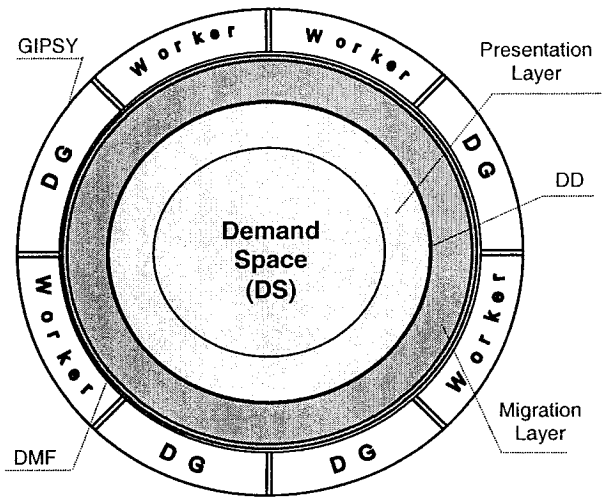


Fig.2.1. GIPSY Demand Migration Framework

The DMF relies on these two functional layers to form an asynchronous communication system similar to the *persistent asynchronous message passing systems* [45], i.e. the messages are delivered on demand, but until that moment, they are kept permanently stored within the system. The messages in the DMF are real objects - demands and results, each result being associated with one demand.

The DD layer establishes the context of a demand-result centralized propagator that consists of two layers - Demand Space (DS) and Presentation Layer (PL) (see Fig.2.1). The DS layer defines the context of an internal object storage mechanism able to store demands and results. The PL is an abstract layer on top of DS, which brings the DS implementation to a generic level.

The Migration Layer makes the heterogeneous distributed communication possible. It is the DMF contact-generic layer with the GIPSY nodes (see Fig.2.1).

Why Layering? As we have seen the DMF establishes a complex model for communication. Therefore, dividing the functionalities into several sets, where the inner functions are tightly coupled, but highly independent from the other layers, is very appropriate. This technique helps to address the problem sets separately and reach for higher upgradeability and flexibility through modularity.

2.2.2. Demand Dispatcher Layer Concerns

The Demand Dispatcher layer establishes a context for maintaining a pool of demands to be processed. The DD is a centralized bootstrap mechanism acting between the DGs and workers. Due to its contributors – Demand Space (DS) and Presentation Layer (PL), the DD is able to keep track of and expose demands and their computed results.

Demand Space Layer. The DD relies on the Demand Space to store all the pending demands and their computed results. The Demand Space layer implies all the characteristics of an Object Database, i.e. the DS provides a mechanism to store the state of objects persistently, and an Object Query Language (OQL) to retrieve these objects. The Object Database Management Group (ODMG) published this standard in 1993 [15].

In our design the DS does not discriminate the demands and results. It maintains them as similar entries - objects. All the entries stored in the Demand Space are permanently saved until they are processed or canceled.

Presentation Layer (PL). The Presentation Layer unifies the functionality of DS to a small set of generic functions, allowing storing, retrieving and canceling demands. The PL exposes the Demand Space transparently, by hiding all the internal DS' details from the public outside and exposing easy-to-use functions. Hence, the Migration Layer simply reads and writes entries - demands and computed results, by relying on the PL's exposed functionality.

At Least Once Delivery Semantics Precautions. The PL takes some extra precautions for uniquely identifying the demands and preventing from an eventual loss of any demand. The PL generates and assigns a Global Unique Identifier (GUID)

to each demand. This GUID becomes a unique demand's signature, thus ensuring *at least once delivery semantics*.

Demands Diversity. The DD discriminates the demands stored in the Demand Space by their processing state, i.e. the demands could be *in process*, *pending* or *computed*. An *in process* demand is one on its way to be delivered to a worker and computed. The DD keeps a copy of that demand until the return of the result of its computation. In that way, the DD prevents eventual loss of a demand, i.e. an *in process* demand could be dispatched again if its result is not received for a certain period of time, thus providing fault-tolerance. The DD removes the copy of any *in process* demand from the Demand Space after its result is received through a *computed* demand that being generated by a worker.

2.2.3. Migration Layer Concerns

The Migration Layer (ML) establishes a context for migrating objects from one node to another. The ML provides a transparent form of migration [4], i.e. the nodes and the objects to be moved do not take special considerations. The ML refers to communication between computers, i.e. it is based on the Open Systems Interconnection (OSI) Reference Model [4, 6]. In addition, the ML provides an architectural structure, forming a multiplatform transport protocol that is able to connect machines with different operating systems. The ML focuses on the use of special kind of *messengers* [2], called Transport Agents (TAs).

Transport Agents Concerns. The Transport Agents are a design solution for the ML model. They are independent components able to carry objects over the machine boundaries. The TAs are mainly responsible for delivering the *pending* demands from the DGs to the DD, *in process* demands from the DD to the workers, and the *computed* demands from the workers back to the DD and from DD back to the DGs.

The Migration Layer context enforces the TAs to expose a common interface to the DD, DGs and workers for demand migration. All the TAs form together the ML transport protocol. Each TA works independently and concurrently with the others. An agent could stop, shutdown or start without affecting the other GIPSY artifacts, thus enabling hot-plugging.

2.3. Applying the DMF

Our DMF is generic. It does not impose technologies or platforms, but guidelines. By applying the DMF, we designed a Demand Migration System (DMS), based on the distributed technologies JINI, CORBA, DCOM and .NET Remoting [7, 13, 14, 16]. Fig.2.2 represents the layered architecture of our DMS derived from the DMF.

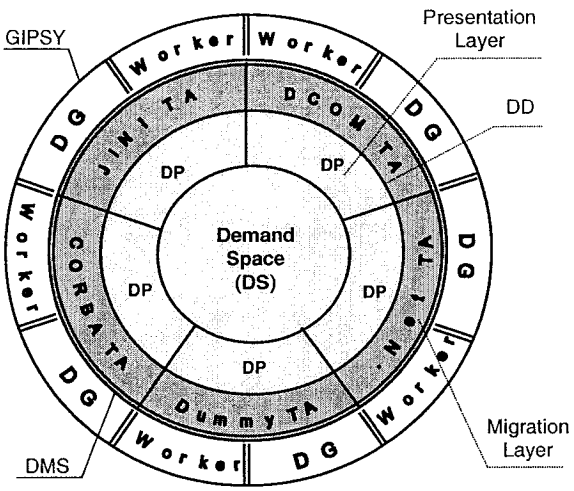


Fig.2.2. GIPSY Demand Migration System

The Demand Dispatcher (DD) and Transport Agents (TA) are subsystems of the DMS, inherited from the DMF. The DD consists of two contributors - Demand Space (DS) and Dispatcher Proxy (DP). Whereas the DS comes directly from the DMF, the DP is a design solution for the Presentation Layer (PL). The PL consists of multiple DPs, each DP being associated with a TA, i.e. a DP is a DD's entry point. The DD has multiple DPs and one single Demand Space. The Migration Layer (ML) is presented as a set of five Transport Agents, each one based on a distributed technology, but the Dummy TA, which simply exposes the DP to the worker and DG. In Fig.2.2 the GIPSY nodes are grouped into pairs, each consisting of a DG and worker that communicate through the DMS. Each DG-worker pair relies on a common TA or on different TAs.

Fig.2.3 represents another abstract conceptual view of our DMS. The largest circle depicts the DMS, and the outstanding rectangles depict DGs and workers, these being the two communication end points, and the DMS acting as a communication intermediate between them. The DD (see the light gray color in Fig.2.3) includes the DS and DPs. The DD has multiple DPs and one single DS. The TAs transport the

demands and results to DGs and workers that adhere to the TAs' interface. In order to communicate with the DD, a TA needs a DP.

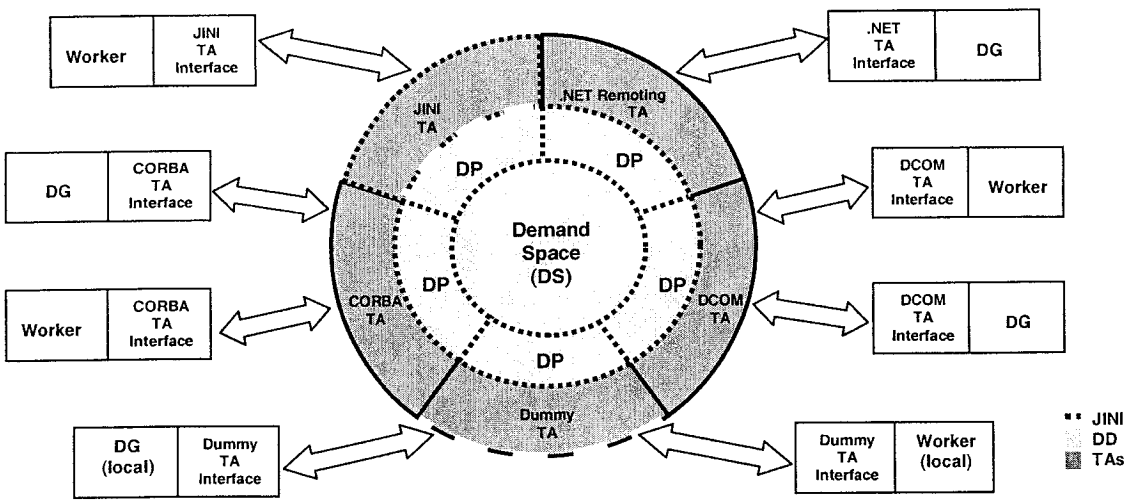


Fig.2.3. GIPSY Demand Migration System as Middleware

In our current implementation, the DMS' core is JINI-oriented (see section 4.). The DMS components – DD and JINI TA, are based on JINI (see the doted line in Fig.2.3). The JINI TA could incorporate the DP, since both are JINI-based (hence the dashed line in the graph between the JINI TA and its corresponding DP).

2.3.1. DMS DD Concerns

The Demand Dispatcher (DD) inherits the DD layer from the DMF. Hence, the DD acts like a middleware between the DGs and workers.

Demand Space (DS). For the demand-result storage mechanism – DS, the DD relies on JavaSpace technology [7] – a JINI implementation that integrates the concept of tuple space [7, 8, 9]. The DD relies on the JavaSpace persistency mechanism to serialize the demands and results, and on the JavaSpace object query mechanism to search for them.

Dispatcher Proxy (DP). The DP inherits the Presentation Layer (PL) from the DMF, i.e. it works as a proxy for the DD. The DD relies on it to expose functionality to its clients. The clients are mostly TAs and some DGs and workers - those related to the local DD case (see Fig.2.4), and each one of them is associated with a unique

DP. The DGs, workers and Transport Agents use the DP functions as their own, in their local address space, which hides the complexity of a possible remote collaboration with the DS.

Local and Remote DD. The distributiveness of DMS architecture exposes two cases of Demand Dispatcher (respectively Demand Space) distribution – local and remote. Local is the case when the Demand Dispatcher is placed locally to the Demand Generator – they both run on the same machine, and there are no machine boundaries to cross in order to connect these two artifacts (see Fig.2.4.), i.e. no TAs are needed. Therefore, the DG should adhere to the dummy TA interface, which simply exposes the Dispatcher Proxy without using any middleware technology (see Fig.2.4). Further, in this document, when we call the DGs, workers or DD local, we refer to this case.

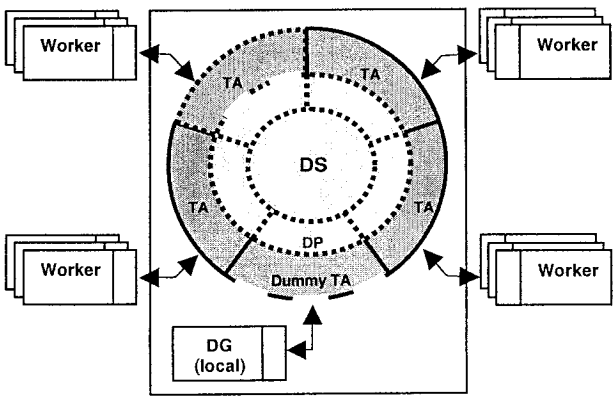


Fig.2.4. Local DD

Remote is the case when the Demand Dispatcher is placed remotely from the DG, i.e. the DD and DG are located on different machines and Transport Agents are needed in order to connect them, i.e. we need to cross the machine boundaries (see Fig.2.5.).

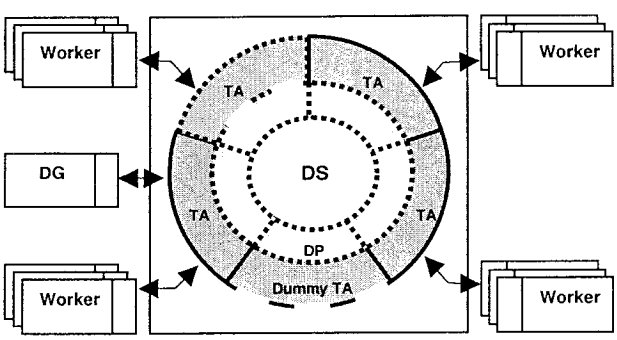


Fig.2.5. Distributed DD

Jagannathan and Dodd talk in [44] about Static Distributed Generator Architecture (SDGA) and Dynamic Distributed Generator Architecture (DDGA), those being models respectively for our Local DD and Distributed DD architecture.

The DG and DD communicate at least twice for each demand to be processed – post a demand and get its result. In addition, a DG could listen constantly to the Demand Space for the required results. Therefore, these two artifacts are tightly tied and dependent on the communication speed. Hence, it is often more appropriate to have these two artifacts grouped together, i.e. the DG and DD run on the same machine. However, in some cases, we might have several DGs using the same DD, or even several DDs, each storing a subset of the demands generated at run-time by the GIPSY. In addition to the two cases investigated above, we could have a case of a local worker and DD and one of a remote worker and DD (see Fig.2.3). These cases are similar to the previous ones. Our DMS architecture is meant to allow any of these variations to allow the dynamic investigation into different run-time architectures.

2.3.2. DMS TA Concerns

In our DMS design the TAs are based on one distributed technology whose architecture influences their implementation. The TAs differ in structure and implementation. Some of them use IDL interfaces [13, 14, 16, 27], others - pure Java interfaces, but they all expose the same interfaces to the DGs, workers and DD. Despite of the distributed technology diversity, the DGs, DD and workers use the Transport Agents transparently.

Transport Agent Interface. When a Transport Agent starts, it plugs into the system by connecting with the DD and exposes its interface to the DGs and workers (see Fig.2.3). Actually, the workers and DGs are the ones who listen for newly plugged TAs and connect to them. The workers and DGs must adhere to the Transport Agent's interface in order to connect to such an agent. Fig.2.2 depicts the DGs and workers on top of a compatible TA.

2.3.3. Multiplatform Transport Protocol Concerns

The requirement for distributiveness is vital for our DMS conception. The DMS initially relies on the distributed technologies JINI, CORBA, DCOM and .NET Remoting [3, 13, 14, 16]. The DMS uses these technologies as a multiplatform transport protocol, able to connect machines with different operating systems (see Fig.2.6). This transport protocol is designed in the form of Transport Agents. Each one of those agents implements the features of only one distributed technology. The transport protocol is open-ended, i.e. we can easily extend it by adding new Transport Agents based on other distributed technologies.

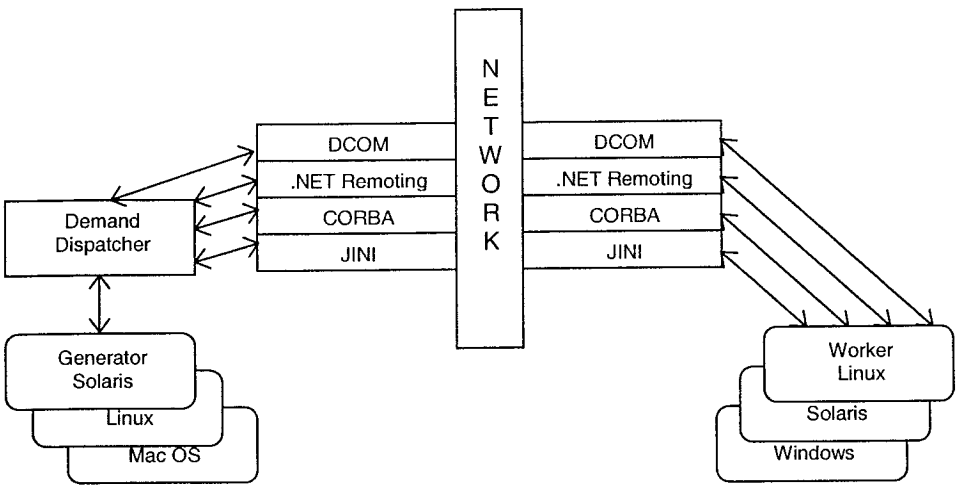


Fig.2.6. Multiplatform Transport Protocol

2.4. Distributed Middleware Concerns

The DMF implies the concepts of distributed systems. A distributed system is a collection of processes/processors that do not share memory or a clock. A distributed application is an application distributed across multiple computers. Some applications by their very nature are distributed because of one or more of the following reasons:

- The data used by the application are distributed.
- The computation is distributed.
- The users of the application are distributed.

2.4.1. Common Foundation of Distributed Technologies

In the course of this thesis, we investigated some of the most advanced distributed technologies, and their implication as distribution middleware for our DMF. We investigated the distributed technologies CORBA, DCOM, .NET Remoting and JINI [3, 7, 13, 14, 16]. All of them, in their core, have a common foundation (see Fig.2.7), based on the Open Systems Interconnection (OSI) Reference Model [4, 6]:

- As computing issue, all of them refer to communication between computers. The combination of software and hardware that allows the computers to communicate is known as a transport layer. By sharing a common transport layer, the computers form a network. The network is the first necessity for all distributed technologies.
- To communicate in a network, the computers need to share the same wire protocol. The wire protocol forms the bottom layer in all distributed technologies.
- Unlike client-server architectures, where the client identifies and communicates directly with the server, the distributed computing architecture introduces the concept of *middleware*. The *middleware* is the functional layer between the client and server. By relying on stubs that run on the client- and server-side, such a layer provides transparency facility to the players in the distributed computing process. The client-side stub is also called *server proxy*. From the DMS perspective, this proxy is called TA Proxy (see section 3.3.1).
- The major participants are clients and servers. The servers presents their services in the system and the clients use them. Therefore, there is a back and forth communication between the participants in a distributed computing process.

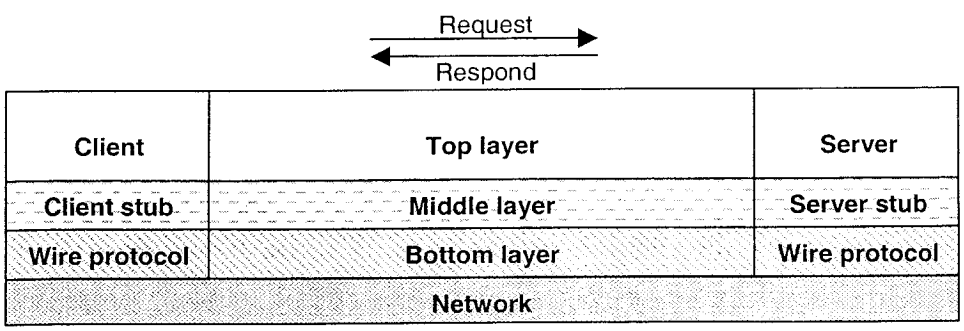


Fig.2.7. Distributed Technologies Foundation

2.4.2. Important Aspects

The following elements reveal some important for our research aspects of distributed middleware.

Distributed Systems Transparency. According to the Colouris [17], a *distributed system* is a collection of autonomous hosts that are connected through a computer network. Each host executes components and operates a distribution middleware. This middleware enables the components to coordinate their activities in such a way that the user perceives the system as a single, integrated computing facility.

From the definition above, we can conclude one of the most important properties of distributed systems – their transparency. In other words, the user of the distributed system shall perceive the system as a single computing facility that internal implementation is hidden by a so-called *abstract layer.*

Migrating Objects to Remote Locations. In the course of this distributed technologies investigation we concentrated our efforts mainly on objects migration. This was one of the principle questions in the course of the DMS design.

Object migration in distributed systems is moving or copying an object from its current host to some other host [4]. If we match this definition to our problem, the object migration is moving or copying demands from a DG to worker and vice versa. The process of object migration is performed by the distribution middleware and is closely related to object persistence.

Communication principles. Another important aspect of our distributed technologies investigation is the communication principles in a distributed environment. These principles helped us to understand the different degrees of reliability with which the process of objects migration can be performed. In addition, we clarified for ourselves the different trade-offs between the reliability and performance, necessitated by the different communication principles.

In general, there are four communication principles followed in a distributed environment [4]. The following elements specify those principles, by referring to the client-server communication depicted in Fig.2.7.

- *Synchronous communication* means that the *client* is blocked while the *server* executes the requested operation. The *client* regains the control only after the *server* has completed the operation or the middleware has notified the *client* about the occurrence of an error.
- *Oneway communication* means the *client* regains the control as soon as the middleware has accepted the request. The *client* and *server* are then executed concurrently and they are not synchronized.
- *Deferred synchronous communication* means that the *client* regains the control as soon as the distribution middleware has accepted the request. The client is not blocked due to the fact that an explicit *request object* is used to get the result. That object performs *synchronous* communication with the *server*. The *client* invokes a function from *request object* to get the result.
- *Asynchronous communication* means that the client regains the control as soon as the distributed middleware has accepted the request. The *server* performs the operation and it explicitly calls a function of the *client* to transfer the result. This technique is called *callback*.

2.5. Summary

In this chapter, we have specified the concepts underlying the generic architecture of our Demand Migration Framework (DMF) and that of the Demand Migration System (DMS), the last being an application of the framework.

The architectural model of our DMF consists of two major contributors called Demand Dispatcher (DD) and Transport Agents (TAs). They both run independently, but form together the overall behavior of the DMF. The DD acts like an event-driven message storage mechanism that uses the TAs to deliver the messages to their recipients. The DMF relies on these two contributors to form a communication system – DMS, similar to the persistent asynchronous message passing systems, i.e. the messages are delivered when they are asked (i.e. in a demand-driven manner), but until that moment, they are kept permanently stored within the message system. The messages in the DMF are demands and results, each result being associated with one demand. The clients of the DMF are Demand Generators (DGs) and workers, both being GIPSY execution nodes.

Chapter 3: Design

> *There are two ways of constructing a software design: one way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies.*
>
> C.A.R. Hoare

This chapter provides a comprehensive architectural overview of the Demand Migration System (DMS) by relying on a number of different architectural views to depict different aspects of the system. It is intended to capture and convey the significant architectural decisions that have been made during the design process.

In our design we followed "4+1" View Model of software architecture [18]. This chapter follows the Software Architectural Document template proposed by the "4+1" View Model. The majority of the sections have been taken from that template.

3.1. Software Architecture

"Software architecture deals with abstraction, with decomposition and composition, with style and esthetics" [18]. To describe the software architecture of the DMS we used the "4+1" model, which is composed of multiple views. We did not use all the five views proposed by the model.

In order to catch most of the functionality and the structural overview of the system, to make the design understandable and easy to use, and finally to catch the non-functional requirements like concurrency, security and availability, we implemented:

- The scenario view, which captures "the most critical functionality of the system" [18] in form of scenarios.
- The logical view as a logical object model of our design. This view could be characterized as a "normal" design view.
- The process view, "which captures the concurrency and synchronization aspects of the design" [18].
- The deployment view, which describes various physical nodes that include process view entities.

3.1.1. Goals and Constraints

Our primary architectural goal was to capture as much as possible of the functional requirements and the most important of the non-functional ones. We form our architecture by considering functional requirements, captured in the scenario, logical, and process views, and nonfunctional requirements, captured in the process and deployment view and supplementary specifications.

We did not consider any constraints imposed by the environment in which the software must operate, by the need for compatibility with existing systems or by the need to reuse existing assets. For our architectural goals we used the preexisting set of architectural principles and policies of the "4+1" View Model and the solutions proposed by multiple design patterns [18, 19, 20]. Our design objective was to design a high-level architecture for the DMS that combines object-oriented programming with distributed computing.

In order to provide an integrated communication system, the architecture has to attain the requirements of concurrency, security, availability, and integrity. Our design does not tackle the performance and efficiency as these are to be tackled by other subsystems of the GIPSY and future subsystems of the DMS.

3.2. Scenario View

This architectural view is "+1" according to "4+1" design view model. This view puts all the parts together. We used a small set of useful and important scenarios to demonstrate how the system works. These scenarios are "in some sense an abstraction of the most important functional requirements" [18]. The scenarios described by this section are instances of more general use cases (see Fig.3.1). Fig.3.1 represents a use case diagram that depicts the use of the DMS by the GIPSY processing nodes DGs and workers, those being presented as actors [40]. The actor "DG" performs "Cancel demand", "Dispatch demand" and "Get result" use cases, and is used by the DMS for performing the "Dispatch result" use case. The actor "worker" performs "Get demand" and "Dispatch result" use cases. In addition, this actor is used by the DMS for performing the "Dispatch demand" use case.

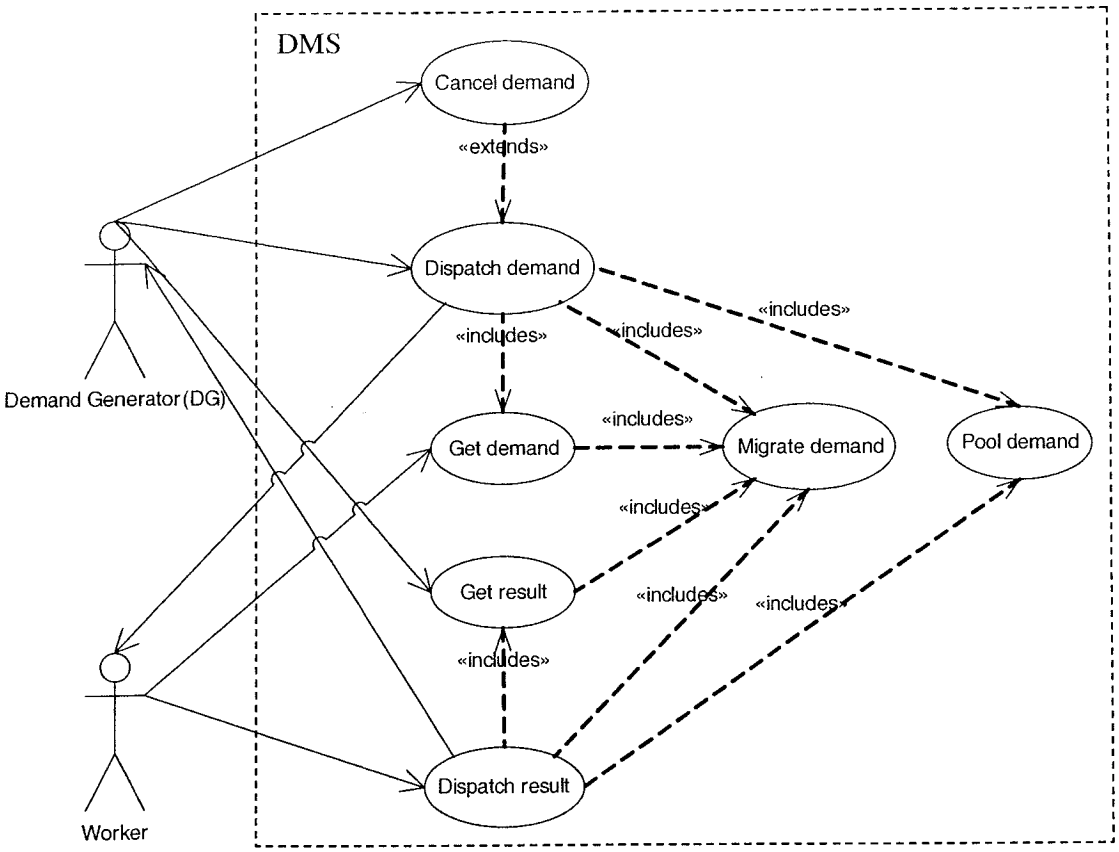


Fig.3.1. DMS General Use Case Diagram

The following scenarios [19, 40] demonstrate the use of the DMS. They are sequences of collaboration activities between the Demand Generator (DG), DMS and worker. The DMS is represented by its components - Transport Agent (TA) and Demand Dispatcher (DD), where the DD is represented by its internal contributors – Dispatcher Proxy (DP) and Demand Space (DS).

3.2.1. The DMS in Use

Before going over the scenarios we illustrate the use of the DMS by giving a simple example. Consider the following formula:

$$D = A + B + C$$

Here A, B and C are complicated formulas from calculus. A DG should compute D, i.e. compute A, B and C first. The DG will generate demands for the computation of A and B, and via the DMS will propagate them to two remote workers, i.e. the DMS will dispatch demands. Each remote worker will execute its demand and generate the corresponding result. Both results will be returned back to the DG via the DMS, i.e.

DMS will dispatch results. Meanwhile, the DG will compute the formula C. Finally, the DG can summarize all the results and compute D.

The following is a scenario for computing the formula "A" by a remote worker. The DD is assumed local for the DG (see section 2.3.1), i.e. no TA is needed for accessing the DD. The sequence diagram [19, 40] in Fig.3.2 depicts this scenario.

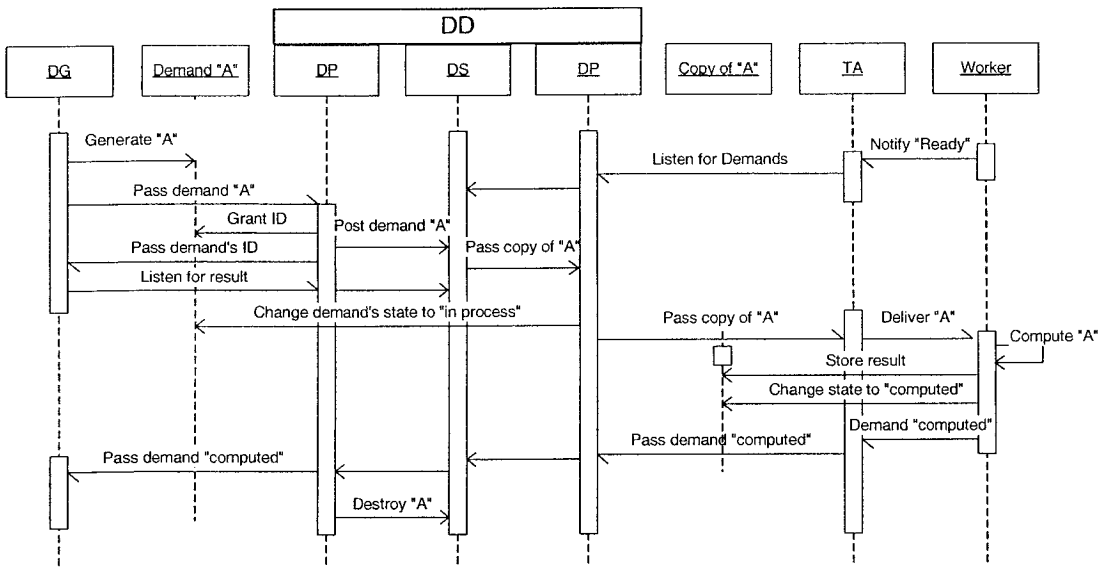


Fig.3.2. Sequence Diagram Formula "A" Computation

Scenario steps:

- A DG generates "A" and passes its computation demand to the DP.
- The DP grants the demand "A" with a unique id (GUID).
- The DP stores the demand in the DS and returns to the DG the demand's id.
- The DG starts listening to the DS for this demand to become *computed*.
- A DP associated with a TA that is listening to the DS for *pending* demands, gets a copy of that demand.
- The DP changes the state of the demand "A" from *pending* to *in process* and sends the copy of "A" to the TA.
- The TA transports the copy of "A" to a *ready* worker.
- The worker (now *busy*) executes the demand, stores the result in the demand, and changes its state to *computed*. The worker reverts to the *ready* state.
- The TA transports the computed demand back to the DD and stores the result in the Demand Space through the Dispatcher Proxy.

- The DD through a Dispatcher Proxy passes the computed demand to the DG and removes the original (*in process*) demand from the Demand Space.

3.2.2. Scenario "Migrate Demand"

This scenario is an instance of the "Migrate Demand" use case (see Fig.3.3). The following table depicts the characteristics of this use case.

Name	Migrate Demand
Actor	None
Description	This use case describes two DMS functions: a) A migration of a demand from a TA to the TA Proxy (see section 3.3.1 for an explanation about the TA Proxy). b) A migration of a demand from a TA Proxy to the TA.
Related use cases	This use case is included by the use cases "Dispatch demand", "Get demand", "Dispatch result" and "Get result".
Pre-condition	Two possible cases: a) A DP associated with a TA holds a demand to be dispatched. b) A GIPSY node (worker or DG) holds a demand to be dispatched.
Base scenario	**Function a)** 1. The DP passes the demand to its associated TA. 2. The TA transports that demand to the TA Proxy, which runs in the GIPSY node's address space. 3. The TA Proxy passes the demand to the GIPSY node (worker or DG). **Function b)** 1. The GIPSY node passes the demand to the TA Proxy, which runs in the GIPSY node's address space. 2. The TA Proxy transports that demand to the TA. 3. The TA passes the demand to the associated DP.
Alternative	None.

Failure	The demand cannot be transported.
Failure conditions	Either the TA Proxy or TA is no longer available due to a network failure, due to a GIPSY node failure, or due to a TA failure.
Post-condition	The demand is transported.

The following sequence diagram [19, 40] depicts the base scenario for function a) of the use case.

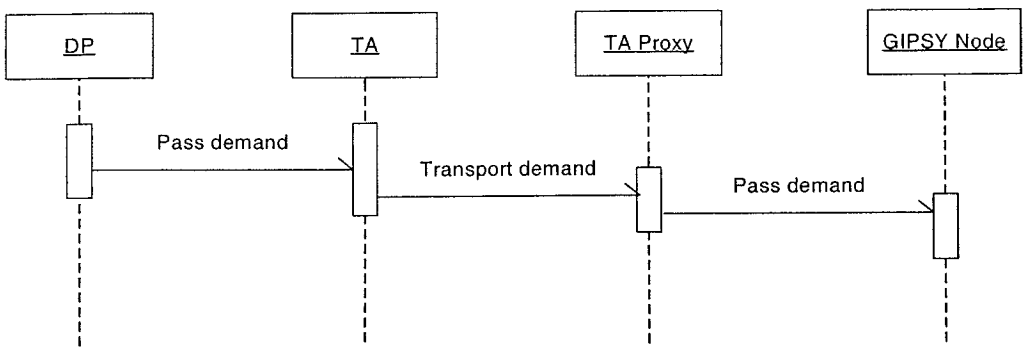


Fig.3.3. Sequence Diagram "Migrate Demand"

The sequence diagram for the demand migration from the TA Proxy to the TA is similar to the diagram in Fig.3.3, but the action is triggered by the GIPSY node.

3.2.3. Scenario "Get Demand"

This scenario is an instance of the "Get Demand" use case (see Fig.3.4). The following table depicts the characteristics of this use case.

Name	Get Demand
Actor	Worker
Description	The worker searches for and gets any *pending* demand from the DS.
Related use cases	This use case is included by the use case "Dispatch demand", and it includes the use case "Migrate demand".
Pre-condition	A pending demand is stored in the DS.

Base scenario	1. The worker completes its work, and makes its associated TA listen to the DS for any *pending* demand. 2. The TA discovers a *pending* demand through its associated DP. 3. The DP changes the status of that demand from *pending* to *in process.* 4. The DP takes a copy of that demand and passes it to the TA. 5. The TA migrates the demand copy to the worker (refers to the use case "Migrate demand").
Alternative	1.1. The DD is local to the worker, and the worker makes its associated DP listen to the DS for any *pending* demand. 1.1.1. The DP discovers a pending demand. 1.1.2. The DP changes the flag of the demand from *pending* to *in process.* 1.1.3. The DP takes a copy of that demand and passes it to the worker.
Failure	1.1. The worker cannot find its associated TA.
Failure conditions	The TA is no longer available due to a network failure or due to a TA failure.
Post-condition	1. The copy of the demand is delivered to the worker. 2. The state of the original demand (kept in the DS) is changed to *in process.*

The following sequence diagram [19, 40] depicts the base scenario of the use case.

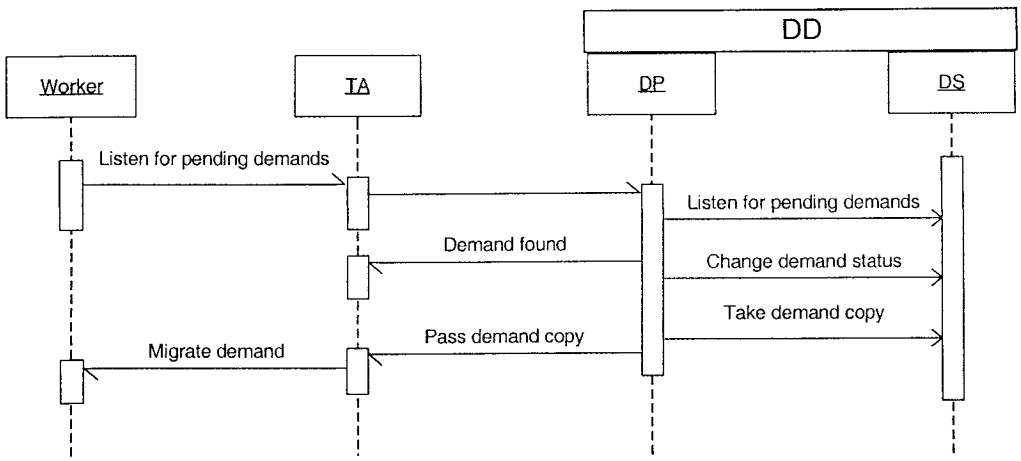


Fig.3.4. Sequence Diagram "Get Demand"

3.2.4. Scenario "Dispatch Demand"

This scenario is an instance of the "Dispatch Demand" use case (see Fig.3.5). The following table depicts the characteristics of this use case.

Name	Dispatch Demand
Actor	DG
Description	The DG generates a demand, and dispatches it to the worker through the DMS. The demand is granted with a GUID at the beginning of the dispatch process.
Related use cases	This use case could be extended by the "Cancel demand" use case. It includes "Pool demand", "Migrate demand" and "Get demand" use cases.
Pre-condition	A newly generated demand.
Base scenario	1. The DG passes the demand to its associated TA (refers to the use case "Migrate demand"). 2. The TA passes the demand to its associated DP. 3. The DP generates a GUID, grants the demand with this GUID, and the TA returns this GUID to the DG. 4. The DG starts listening to the DS for this demand to become *computed*. 5. Meanwhile, the DP stores the demand in the DS (refers to

	the use case "Pool Demand"). 6. A *ready* worker gets a copy of that demand through its associated TA (refers to the use case "Get Demand").
Alternative	1.1. The DD is local to the DG, and the DG passes the demand to its associated DP. 1.1.1. The DP grants the demand with a GUID and returns this id to the DG. 1.1.2. Continue with 4. 4.1. The DG cancels the demand (refers to the use case "Cancel demand".
Failure	1.1. The DG cannot pass the demand to its associated TA.
Failure conditions	The TA is no longer available due to a network failure or due to a TA failure.
Post-condition	The demand is delivered to the worker.

The following sequence diagram [19, 40] depicts the base scenario of the use case.

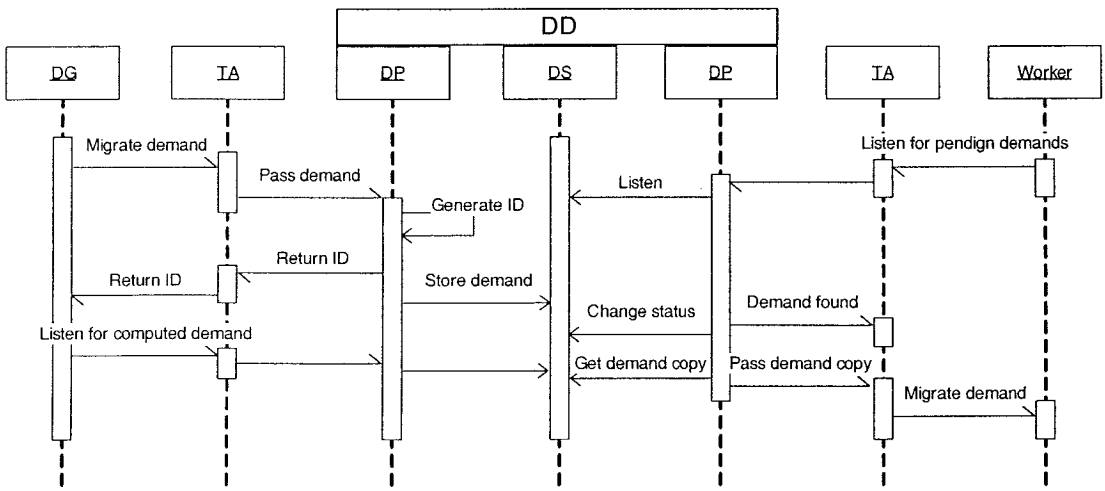


Fig.3.5. Sequence Diagram "Dispatch Demand"

3.2.5. Scenario "Cancel Demand"

This scenario is an instance of the "Cancel Demand" use case (see Fig.3.6). The following table depicts the characteristics of this use case.

Name	Cancel Demand
Actor	DG
Description	The DG requests from the DMS to cancel a demand dispatched for computation by providing the demand's GUID. Demands in any state – *pending*, *in process* or *computed*, can be canceled.
Related use cases	This use case extends the use case "Dispatch Demand".
Pre-condition	A demand has been dispatched for computation by the DG.
Base scenario	1. The DG sends a "cancel demand" message to the TA with the ID of the demand to be canceled. 2. The TA redirects that message to its associated DP. 3. The DP deletes the demand from the DS.
Alternative	None.
Failure	The cancel message cannot be delivered to the TA.
Failure conditions	The TA is no longer available due to a network failure, or due to a TA failure.
Post-condition	The demand is canceled (deleted from the DS).

The following sequence diagram [19, 40] depicts the base scenario of the use case.

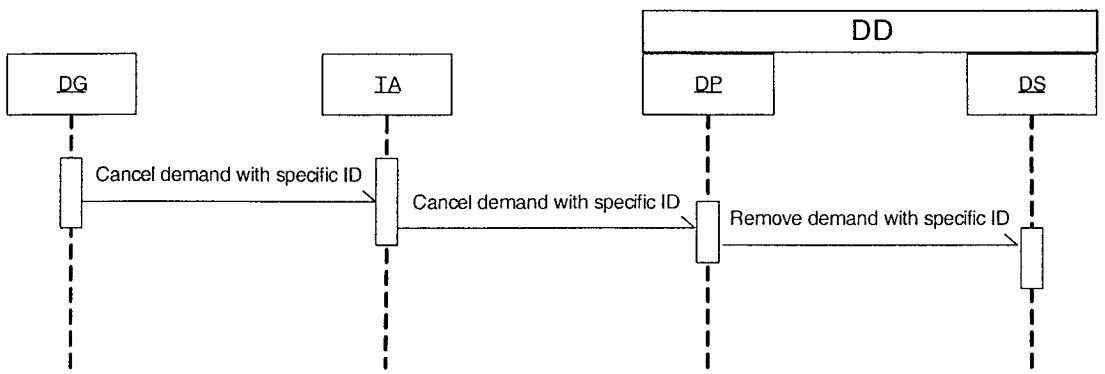


Fig.3.6. Sequence Diagram "Cancel Demand"

3.2.6. Scenario "Pool Demand"

This scenario is an instance of the "Pool Demand" use case (see Fig.3.7). The following table depicts the characteristics of this use case.

Name	Pool Demand
Actor	None.
Description	The DD pools demands. The DP wraps demands and their state in an entry format to be stored in the DS, and stores those entries in the DS. Only *pending* and *computed* demands can be pooled, since the *in process* demand is a demand already stored in the DS, but with a changed state.
Related use cases	This use case is included by the use cases "Dispatch Demand" and "Dispatch Result".
Pre-condition	The DP holds a demand to be stored in the DS.
Base scenario	1. The DP creates an entry object. 2. The DP adds the demand to this object with the appropriate state – *pending* or *computed*. 3. The DP stores the entry object in the TS.
Alternative	None.
Failure	The DP cannot store the entry object in the DS.
Failure conditions	The DS is no longer available due to a DS failure or due to insufficient storage space.
Post-condition	The demand is pooled (stored in the DS).

The following sequence diagram [19, 40] depicts the base scenario of the use case.

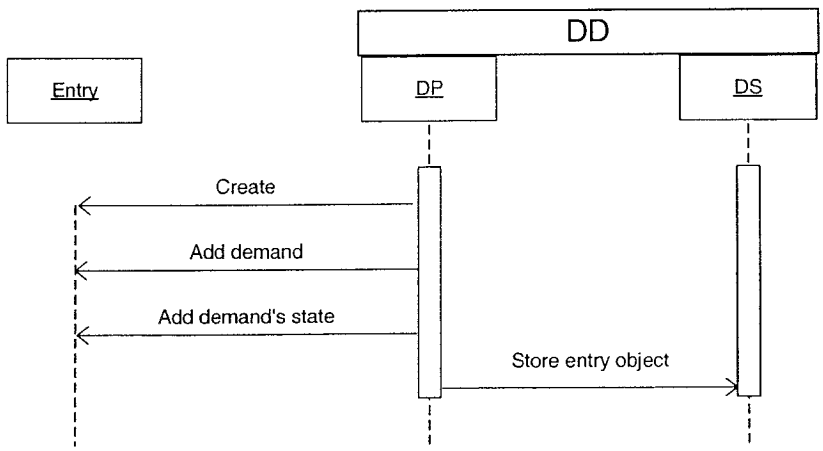


Fig.3.7. Sequence Diagram "Pool Demand"

3.2.7. Scenario "Get Result"

This scenario is an instance of the "Get Result" use case (see Fig.3.8). The following table depicts the characteristics of this use case.

Name	Get Result
Actor	DG
Description	The DG searches for and gets a *computed* demand from the DS. The *computed* demand is discovered by its GUID.
Related use cases	This use case is included by the use case "Dispatch result".
Pre-condition	A computed demand with a specific GUID is stored in the DS.
Base scenario	1. The DG listens to the DS via its associated TA for a *computed* demand with a specified GUID. 2. The TA discovers the *computed* demand through its associated DP. 3. The DP takes that demand, and passes it to the TA. 4. The TA migrates the *computed* demand to the DG (refers to the use case "Migrate Demand").
Alternative	1.1. The DD is local to the DG, and the DG makes its associated DP listen to the DS for a *computed* demand with a specified GUID. 1.1.1. The DP discovers the computed demand. 1.1.2. The DP takes that demand, and passes it to the DG.
Failure	1.1. The DG cannot find its associated TA.
Failure conditions	The TA is no longer available due to a network failure or due to a TA failure.
Post-condition	1. The *computed* demand is delivered to the DG. 2. The computed demand is removed from the DS.

The following sequence diagram [19, 40] depicts the base scenario of the use case.

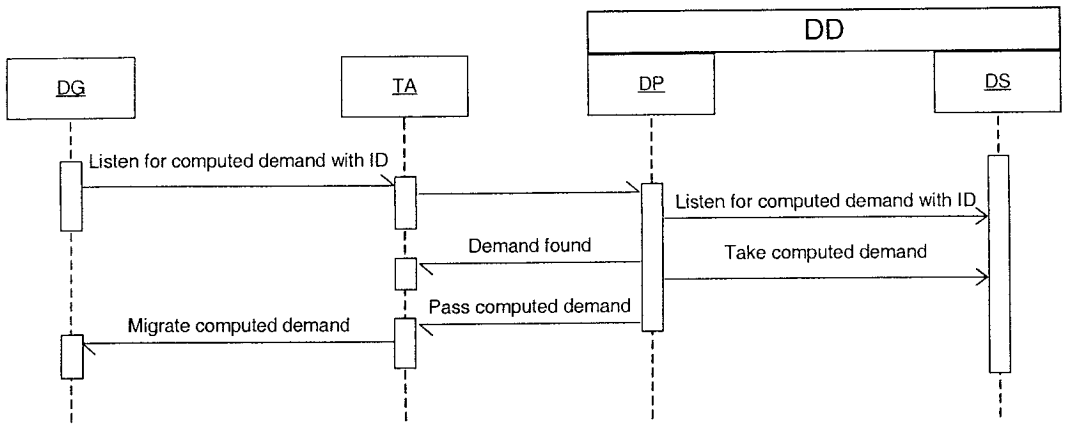


Fig.3.8. Sequence Diagram "Get Result"

3.2.8. Scenario "Dispatch Result"

This scenario is an instance of the "Dispatch Result" use case (see Fig.3.9). The following table depicts the characteristics of this use case.

Name	Dispatch Result
Actor	Worker
Description	The worker computes the received demand, generates a *computed* demand with the computation result, and dispatches the *computed* demand back to the DG via the DMS. The *computed* demand has the same GUID as its mentor (the received demand).
Related use cases	This use case includes "Pool Demand", "Migrate Demand" and "Get Result" use cases.
Pre-condition	A worker has received a demand for computation.
Base scenario	1. The worker computes the received demand. 2. The worker generates a *computed* demand, holding the computation result and the same GUID as the received demand. 3. The worker passes the demand to its associated TA (refers to the use case "Migrate Demand"). 4. The TA passes the demand to its associated DP.

	5. The DP stores the demand in the DS (refers to the use case "Pool Demand"). 6. The worker starts listening to the DS for any *pending* demand. 7. A DG gets the stored *computed* demand through its associated TA (refers to the use case "Get result").
Alternative	3.1. The DD is local to the worker, and the worker passes the demand to its associated DP. 3.1.1. Continue with 5.
Failure	1.1. The worker cannot pass the demand to its associated TA.
Failure conditions	The TA is no longer available due to a network failure or due to a TA failure.
Post-condition	The *computed* demand is delivered to the DG.

The following sequence diagram [19, 40] depicts the base scenario of the use case.

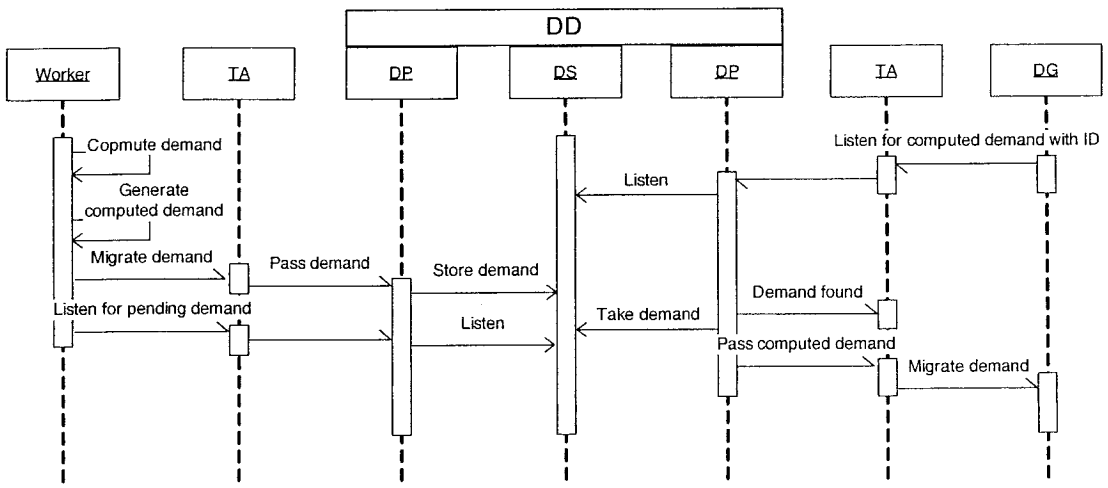


Fig.3.9. Sequence Diagram "Dispatch Result"

3.3. Logical View

In our design the "logical view is the object model of the design (when an object oriented design method is used)" [18]. It primarily supports the functional requirements - "what the system should provide in terms of services to its users" [18]. We used UML [40] in terms of static structure elements – classes and packages to

build the Demand Migration System logical architecture view (see Fig.3.10). The following diagram is an overview model that represents the major subsystems as software packages [19, 40]. A package here is a collection of classes that are logically grouped together. In addition, the diagram depicts the packages' interface(s) and the associations among the packages. The interfaces are depicted as packages' entry point(s), i.e. the communication among the different packages is possible only through those interfaces.

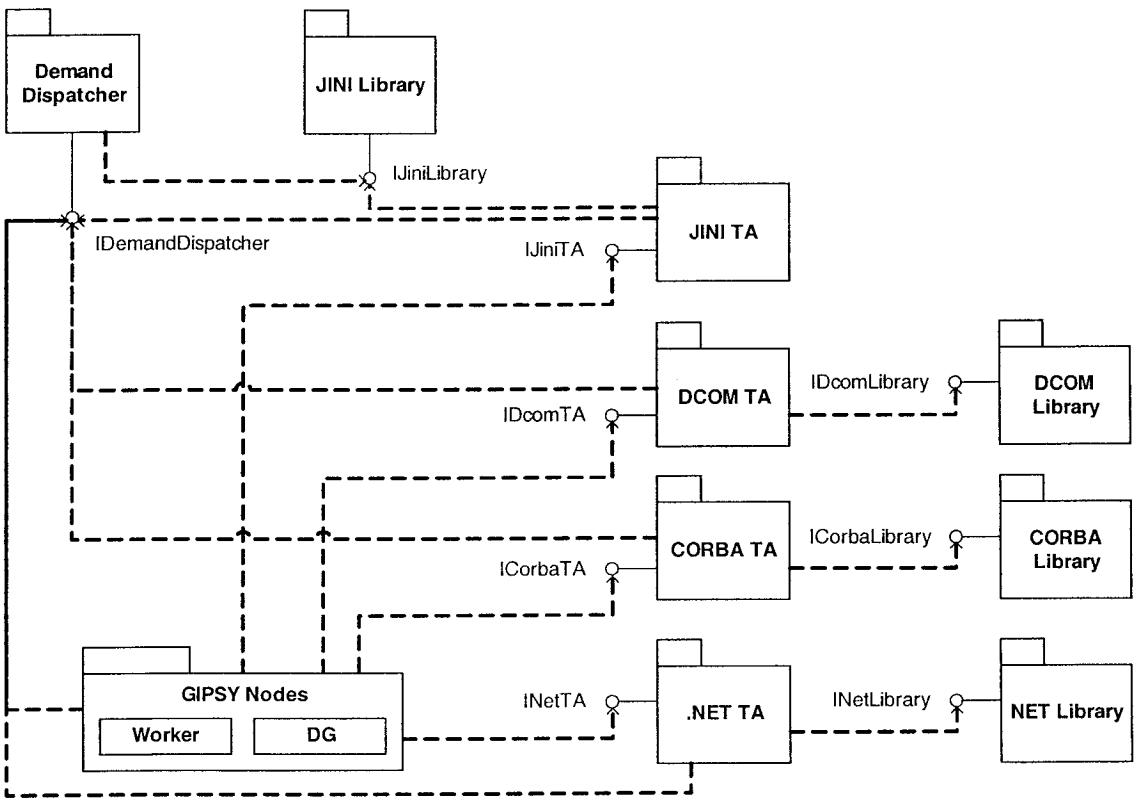


Fig.3.10. Overall Logical Architecture View

3.3.1. Design Rationale

The Design Rationale (DR) for the overall system expresses elements of the reasoning, which has been invested in the design of the whole system. "A DR answers *"Why...?"* questions of different sorts, depending on the class of DR represented" [19]. As an answer to the requirements of the DMS we propose nine major modules – packages. Most of them are derived by the DMS conceptual architectural view (see section 2.3). The GIPSY Nodes package is not a part of the

DMS architecture. It is presented in the logical view in order to achieve a complete overall relationship view. Those modules group logically different system artifacts – classes. The following is a short description of some principle questions that rised during the design process and their logical answers implemented in our design.

Demand Dispatcher. The Demand Dispatcher package incorporates the functionality of the DD layer (see section 2.2 and section 2.3). This package exposes an interface called *IDemandDispatcher* with all the accessible generic package's functions. The package is used by the GIPSY Nodes package and by the TA packages, each one being interested in the *IDemandDispatcher* interface (see Fig.3.10). For performing its functionality, the Demand Dispatcher package relies on the JINI Library package. The Demand Dispatcher package encapsulates the Demand Space (DS) and the Presentation Layer (PL) together.

Why the DS and PL are encapsulated in one package? In our architecture, the DS and PL are tightly related to each other. The PL presents the DS, i.e. the PL is the entry point to the DS. Therefore, the DS is an inner component of the DD, i.e. the DS cannot expose its own public interface accessible by the other packages.

TA packages. The TA packages depict the Transport Agents. There are four TA packages – JINI TA, CORBA TA, DCOM TA and .NET TA, each one being related to a corresponding distributed computing technology [7, 13, 14, 16]. The TA packages expose their functionality through a TA interface. All the TA packages are interested in the Demand Dispatcher package, and are used by the GIPSY Nodes package (see Fig.3.10), since they are the communication layer between the DD and GIPSY nodes (see section 2.3). In our design, every TA is designed in a single package.

Why a single package for every TA? The TAs are designed as stand-alone components, independent from each other. In addition, each TA is based on a distributed technology (see section 4.1) whose architecture influences the TA design and implementation. Therefore, the TAs differ in design structure and implementation.

TA stubs. The TAs work in a distributed environment. This requires a stub generation (see section 2.4.1). Therefore, in our design we should take into consideration the TA stubs, which we called TA Proxies.

Distributed technologies libraries. In our design, the distributed technologies are encapsulated in single packages. These packages encapsulate only the distributed technology functionality needed by the DMS and expose this functionality in a generic manner through their interface (see Fig.3.10). We designed four packages – JINI Library, DCOM Library, CORBA Library and .Net Library, each one being associated with one of the distributed technologies. The TA packages are interested in the packages listed above and use them through their interface.

The rest of the logical view's subsections are made up of a package name. For the three packages JINI Library, Demand Dispatcher and JINI TA there is a design rationale and detailed design, including a UML class diagram [19, 40] and description of classes that make up that package. Following the definition of any class is a list and definition of the methods and data fields that make up that class.

3.3.2. JINI Library

The JINI Library package implements all the functionality necessary for using JINI as a heterogeneous distributed computing middleware. The package generalizes JINI and brings its functionality to a higher level of abstraction, those being ensured by the package's interface *IJINILibrary*. The JINI Library package is used by all the DMS' components based on JINI. Such components are the JINI TA and an implementation of the DS based on JavaSpace. The following UML class diagram depicts the JINI Library package's classes and their relationships.

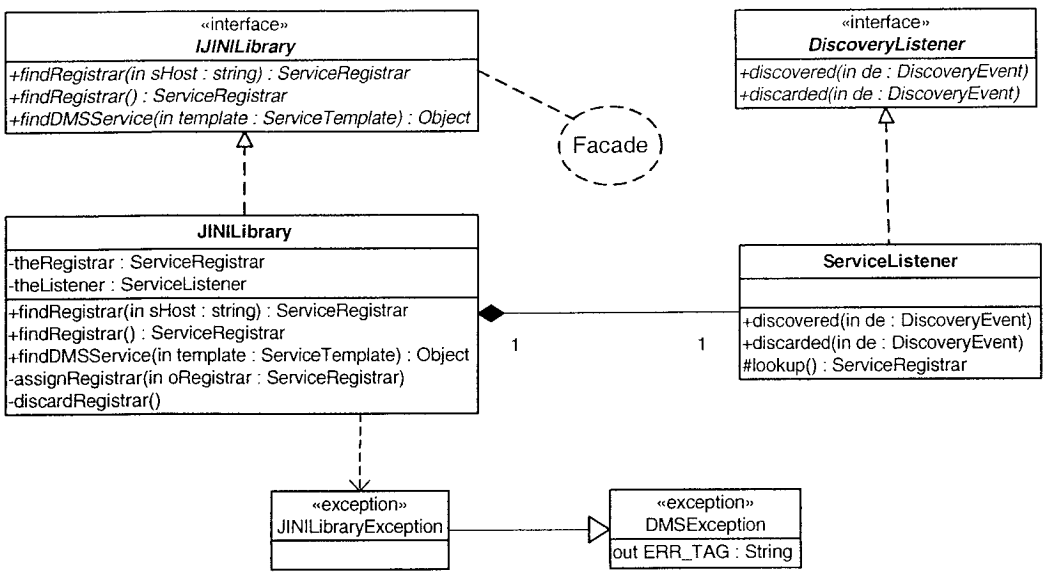


Fig.3.11. JINI Library Class Diagram

3.3.2.1. Design Rationale

The following is a list of questions that raised during the design process of the JINI Library package and their reasonable answers implemented in our design.

The JINI Library is an abstraction of some JINI functionality. All the necessary JINI functionality, needed by the DMS' components, is exposed by a set of generic functions through the *IJINILibrary* interface. In our design process, we applied the *Facade* pattern [19, 20], which brings the package's functionality to a higher level of abstraction (see Fig.3.11). This solution was derived from the layered architectural structure exposed by the DMS. I.e. the implementation of the technology related functionality in a different package gets along pretty well with the high packages independency. JINI is a complex environment (see section 4.1.1) where the different services required different approach. Therefore, it is better to implement those approaches in a transparent manner. An *alternative solution* was to implement the needed JINI functions at the DMS class level. Since our DMS does not rely on all the JINI functionality, but on a small set of JINI functions, this approach was investigated as well. The problems raised by this approach were mainly related to the contradiction with the generic architectural structure of the DMS and its interoperability. The last one as a nonfunctional requirement is coming from the ability of DMS to work with different distributed computing technologies.

Multicast and unicast discovery. There are two different ways of discovering JINI services [3, 7, 34] – multicast and unicast discovery (see section 4.1.1.2). The DMS' JINI related components are designed as JINI services. Hence, the service discovery mechanisms are some of the important features we should implement in our JINI Library. These features are directly exposed by the package's interface.

Registrars. All the JINI services must be registered with a registrar [3, 7, 34]. Hence, our DMS JINI services must be registered with such a registrar as well.

Service listener. The service listener is a mechanism for discovering JINI services in an asynchronous manner. Our investigation into the JINI distributed events, which are the mechanism used for asynchronous communication in the JINI systems, helped to design a service listener mechanism. This mechanism helps for *late discovering*. We designed the service listener as an inner class for the implementer of the *IJINILibrary* interface.

Why is the service listener an inner class? The use of an inner class is appropriate here, because it is mainly responsible for handling events. Hence, an instance of this class works in a background mode and calls methods from the main class, when an event occurred. This is most appropriate for the asynchronous service discovery, i.e. the main class does not block due to a possibly long discovery process.

Why late discovery? Since the DMS nodes are "volunteers" (see section 2.1), .i.e. they are designed to allow "hot-plugging", the *late discovery* helps in discovering services (DMS' JINI related components) that plug into the DMS later.

3.3.2.2. Detailed Design

The following is a description of the JINI Library package's classes and their attributes and methods.

Interface *IJINILibrary*

The *IJINILibrary* interface is the package's entry interface. It exposes three generic functions that make the JINI Library package easier to use. All the interface's methods throw a *JINILibraryException* exception. The following is a description of these methods and their pseudo code.

Method *findRegistrar()*

The interface exposes two overloading *findRegistrar* methods. These methods find a JINI service registrar that is needed for locating the desired DMS service.

The first method accepts as a parameter the host address and performs a unicast discovery process [3, 7].

The second one has no parameters and performs multicast discovery process [3, 7].

Pseudo code

```
public ServiceRegistrar
  findRegistrar (String sHost);

public ServiceRegistrar
  findRegistrar ();
```

Method *findDMSService()*

This method uses the discovered by one of the previous methods registrar to locate the desired DMS service. A service template passed as a parameter to the method describes the desired service.

Pseudo code

```
public Object findJiniService (
  ServiceTemplate template);
```

Class *JINILibrary*

The *JINILibrary* class is the main class for the package. It implements the *IJINILibrary* interface, i.e. it implements all the necessary functions for finding a registrar and locating the desired DMS service. The class *JINILibrary* implements an inner service listener class (see the *ServiceListener* class below), which is used by the multicast discovery mechanism implementation. The class *JINILibrary* implements the following methods and attributes.

Data field *theRegistrar*

This private data field is designed to keep the last discovered registrar.

Pseudo code

```
private ServiceRegistrar theRegistrar;
```

Data field *theListener*

This private data field holds a reference to the service listener.

Pseudo code

```
private ServiceListener  theListener;
```

Method *findRegistrar()*

There are two overloading *findRegistrar* methods implemented by the class. Those methods implement the *findRegistrar* methods of the *IJINILibrary* interface (see the *IJINILibrary* interface).

Pseudo code

```
public ServiceRegistrar findRegistrar (
String sHost) {
   Perform a unicast discovery to find the
   registrar on the host=sHost.
   Return the registrar. }
public ServiceRegistrar findRegistrar () {
   Perform a multicats discovery to find any
   registrar all around the network, by using
   theListener.
   Return the registrar.}
```

Method *findDMSService()*

This method implements the method *findDMSService* of the *IJINILibrary* interface (see the *IJINILibrary* interface).

Pseudo code

```
public Object findDMSService ( ServiceTemplate
template) {
 if theRegistrar=null then
    call findRegistrar.
 Look for the DMS service in theRegistrar by
 matching the service template.}
```

Method *assignRegistrar()*

This method assigns the discovered registrar to the data filed *theRegistrar*. The method is called by the *findRegistrar* methods and by the inner class *ServiceListener*.

Pseudo code

```
private void assignRegistrar ( ServiceRegistrar
oRegistrar)
{
   if (theRegistrar==null)
    theRegistrar = oRegistrar
}
```

Method *discardRegistrar()*

This method assigns to the data filed *theRegistrar* a null value.

Pseudo code

```
private void discardRegistrar () {
    theRegistrar = null}
```

Interface *DiscoveryListener*

The interface *DiscoveryListener* is a JINI interface implemented by the class *ServiceListener* (see the *ServiceListener* class below). The interface defines two methods – *discovered* and *discarded*.

Method *discovered()*

This method receives a discovery event when one or more registrars are discovered. This method is called by JINI when the desired registrar is found.

Pseudo code

```
public void discovered(DiscoveryEvent ev);
```

Method *discarded()*

This method is called by JINI only when we explicitly discard the registrar, i.e. when the registrar goes down.

Pseudo code

```
public void discarded(DiscoveryEvent ev);
```

Class *ServiceListener*

The *ServiceListener* class is an inner class for the class *JINILibrary*. It implements the *DiscoveryListener* interface. The class is capable to catch the discovery events occurring when a registrar is discovered or discarded. The *JINILibrary* class relies on this class for performing a multicast discovery process. The class *ServiceListener* implements the following methods:

Method *discovered()*

This method implements the *discovered* method of the *DiscovoryListener* interface (see the *DiscovoryListener* interface).

Pseudo code

```
public void discovered(DiscoveryEvent ev) {
  if a registrar is discovered then
  call the assignRegistrar method of the
  JINILibrary class in order to assign the
  discovered registrar }
```

Method *discarded()*

This method implements the *discarded* method of the *DiscovoryListener* interface (see the *DiscovoryListener* interface).

Pseudo code

```
public void discarded(DiscoveryEvent ev) {
  if the registrar is discarded then
  call the discardRegistrar method of the
  JINILibrary class
}
```

Exceptions

In our design solution for the JINI Library package, we defined an exception called *JINILibraryException*. It inherits the *DMSException* (see Fig.3.11) and encapsulates all the possible JINI Library exceptions. The *JINILibraryException* exception can be thrown by the methods of the *JINILibrary* class.

3.3.3. Demand Dispatcher

The Demand Dispatcher (DD) package implements the functionality of the demand propagator (see section 2.2.1). It consists of two major functional layers – the Demand Space (DS) and Presentation Layer (PL). Whereas the DS stores and queries the demands the PL generalizes the functionality of the DS, by exposing it in

a transparent manner. The following UML class diagram depicts the DD package's classes and their relationships.

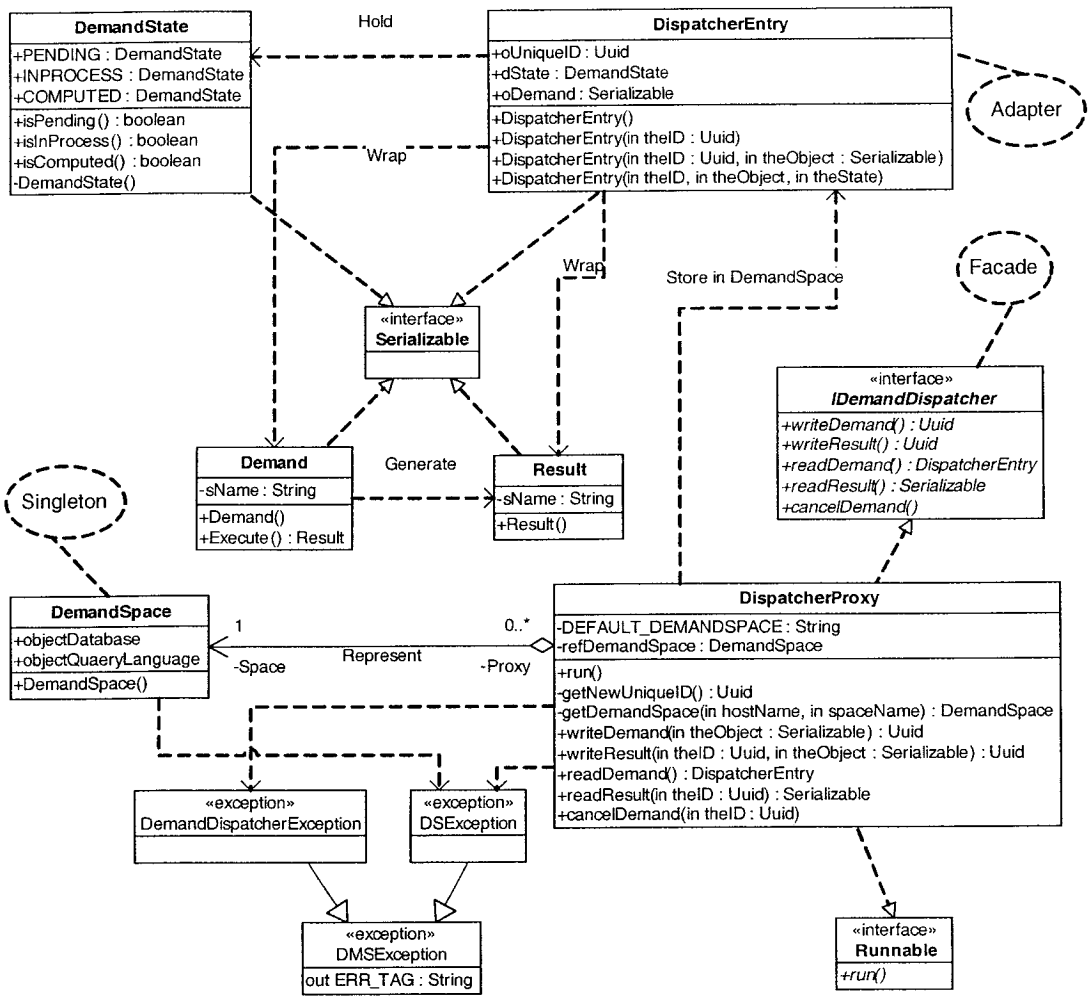


Fig.3.12. Demand Dispatcher Class Diagram

3.3.3.1. Design Rationale

The following is a list of questions that raised during the design process of the DD package and their reasonable answers implemented in our design.

Dispatcher Proxy (DP). The PL is the abstract layer on top of the DS used by the TAs, i.e. it should play the role of a proxy for the DS with the TAs. The need of an abstraction layer, which fulfills presentation functions, necessitates the use of the *Facade* pattern [19, 20], which makes the subsystem easier to use. By applying this pattern, we succeeded in designing a higher-level interface and its implementer class

called Dispatcher Proxy (DP) (see Fig.3.12). The PL is a set of DPs, each one being associated with a TA. This reduces the concurrency to the level of the DS.

Demand Space (DS). The DMS must be able to store and retrieve demands. Hence, it should implement a data storage mechanism able to store demands and a demand query mechanism able to retrieve the stored demands.

Demands unification. The DD deals with three kinds of demands – *pending*, *in process* and *computed*. The DS should not discriminate them, i.e. it should unify them in terms of storage units. For this purpose, we applied the *Adapter* pattern [19, 20], which converts the interface of a class into another interface clients expect, i.e. it wraps the demands and results in an entry format. Our solution to this pattern issue is the *DispatcherEntry* class (see Fig.3.12).

Why is the unification appropriate here? The DD maintains the demands equally without any internal distinction. Hence, the demands unification surely brings simplification, which makes consistent the demand storing, retrieving and querying. In addition, the use of an adapter is valuable, due to the fact it provides new functionality the adapted class does not provide. For example, the demand state and Global Unique Identifier (GUID) (see section 3.3.3.2 for the *DispatcherEntry* class).

Data persistency. The demands must be able to survive the lifetime of the process, i.e. they should be persistent. Hence, they will survive failures and host shutdowns [4]. In Java, object persistency is provided by the *Serializable* interface [23]. Hence, our demands and their results should adhere to the *Serializable* interface. Since the *DispatcherEntry* class is the DS storage entry, it should adhere to this interface as well.

Why a singleton Demand Space? The Demand Space is a *Singleton* [19, 20]. The use of the *Singleton* pattern assures global access to the DS from all the Dispatcher Proxies (respectively from all the TAs). Such a solution should take in consideration concurrent access, since multiple TAs should be able to access the DS simultaneously (for more about concurrency see section 3.4.1). In terms of performance it is not the best solution. An *alternative solution* is to design the DS

able to instantiate multiple instances, each instance being associated with a TA. But, it is difficult to get it along with the necessity to access the DS from all the TAs. Also, such design will lead to less memory efficiency – multiple copies of the DS will require a lot more memory than a single one.

3.3.3.2. Detailed Design

The following is a description of the Demand Dispatcher package's classes and their attributes and methods.

Interface *IDemandDispatcher*

The *IDemandDispatcher* interface is the package's entry interface. It exposes the package's functionality, by relaying on the methods, described below. All the methods exposed by the interface throw the *DemandDispatcherException* exception.

Method *writeDemand()*

This method writes a *pending* demand to the Demand Space. In addition, it generates the GUID for the demand and returns this GUID as a result.

Pseudo code

```
public Uuid writeDemand(
  Serializable theObject)
  throws DemandDispatcherException;
```

Method *writeResult()*

This method writes a result (*computed* demand) to the Demand Space. It assigns the demand's GUID to its result.

Pseudo code

```
public Uuid writeResult(
  Uuid theID, Serializable theObject)
  throws DemandDispatcherException;
```

Method *readDemand()*

This method reads a *pending* demand from the Demand Space.

Pseudo code

```
public DispatcherEntry readDemand()
  throws emandDispatcherException;
```

Method *readResult()*

This method reads a result (*computed* demand) from the Demand Space, having certain GUID.

Pseudo code

```
public DispatcherEntry readResult(
  Uuid theID)
  throws DemandDispatcherException;
```

Method *cancelDemand()*

This method cancels any demand, despite of its status, i.e. removes the demand from the Demand Space.

Pseudo code

```
public void cancelDemand(Uuid theID)
  throws DemandDispatcherException;
```

Class *DispatcherProxy*

The *DispatcherProxy* class is the design solution for the Presentation Layer. It implements the *IDemandDispatcher* interface, i.e. all the necessary functions for reading, writing and canceling demands. In addition, this class implements the *Runnable* Java interface [23] (allowing execution in a separate thread of control) and functionality for connecting the DS and open and close DS sessions.

Method *writeDemand()*

This method implements the *writeDemand* method of the *IDemandDispatcher* interface (see the *IDemandDispatcher* interface).

Pseudo code

```
public Uuid writeDemand(
  Serializable theObject)
  throws DemandDispatcherException {
Open a DS session.
Generate the GUID.
Create a DispatcherEntry object.
Assign the demand and GUID to this object.
Store the DispatcherEntry object in the DS.
Close the DS session and return the GUID.}
```

Method *writeResult()*

This method implements the *writeResult* method of the *IDemandDispatcher* interface (see the *IDemandDispatcher* interface).

Pseudo code

```
public Uuid writeResult(
  Uuid theID, Serializable theObject)
  throws DemandDispatcherException {
Open a DS session.
Create a DispatcherEntry object.
Assign the result and GUID to this object.
Store the DispatcherEntry object in the DS.
Close the DS session and return the GUID.}
```

Method *readDemand()*

This method implements the *readDemand* method of the *IDemandDispatcher* interface (see the *IDemandDispatcher* interface).

Pseudo code

```
public DispatcherEntry readDemand()
  throws emandDispatcherException {
Open a DS session.
If there is an entry stored in the DS and if its
    state=pending then get a copy of this entry.
Close the DS session and return the copy.}
```

Method *readResult()*

This method implements the *readResult* method of the *IDemandDispatcher* interface (see the *IDemandDispatcher* interface).

Pseudo code

```
public DispatcherEntry readResult( Uuid theID)
  throws DemandDispatcherException {
Open a DS session.
If there is an entry stored in the DS and if its
    state=computed and if its GUID = theUD
    then get a copy of this entry.
Close the DS session and return the copy.}
```

Method *cancelDemand()*

This method implements the *cancelDemand* method of the *IDemandDispatcher* interface (see the *IDemandDispatcher* interface).

Pseudo code

```
public void cancelDemand(Uuid theID)
  throws DemandDispatcherException {
Open a DS session.
If there is demand stored in the DS and if its
    GUID=theID then delete demand.
```

Close the DS session.}

Method *getDemandSpace()*

This method gets a reference to the Demand Space object, located on the local or remote machine (specified by its IP address). The DS reference is used by all the operations requiring interaction with the DS. In addition, the method secures the DS context, by taking some security connection precautions for the remote connection.

Pseudo code

```
private DemandSpace getJavaSpace(
  String hostName, String spaceName)
{
 If there is a DS run on the host = hostname and
 if the DS name = spaceName then
    get the reference to that DS.
 Establish a security connection context.
 Return the DS reference.
}
```

Data field *refDemandSpace*

This data field holds a reference to the connected Demand Space.

Pseudo code

```
private  DemandSpace refDemandSpace;
```

Constructor *DemandDispatcher()*

This method is the constructor of the class. It calls the *getDemandSpace* method and assigns the returned DS reference, if any, to the *refDemandSpace* data field (variable). If the returned DS reference is null, the constructor raises an exception, i.e. the constructor succeeds if and only if there is an established connection to the DS.

Pseudo code

```
public DemandDispatcher(
   String hostName, String spaceName)
throws DemandDispatcherException
{
 Call getJavaSpace with the parameters
 hostname and spaceName and assign the
 result to the refDemandSpace.
 If refDemandSpace = null then
    throws DemandDispatcherException
}
```

Class *DemandSpace*

The class *DemandSpace* is our design solution for the Demand Space. This class is *Singleton* [19, 20], i.e. it could instantiate only one instance. The class implements two major functionalities - a demand storage mechanism and demand query mechanism. Since we did not intend to design and implement an Object Database with the appropriate Object Query Language [15], the class was designed to integrate some already existing ones. Hence, in our design solution the *DemandSpace* class holds two public data fields referencing to a demand storage mechanism and demand query mechanism, those being part of the integrated Object Database.

Data field *objectDatabase*

This data field holds a reference to an Object Database able to store demands wrapped as Dispatcher Entries.

Pseudo code

```
public Object objectDatabase;
```

Data field *objectQueryLanguage*

This data field holds a reference to an Object Query Language able to query the Object Database for demands wrapped as Dispatcher Entries.

Pseudo code

```
public Object objectQueryLanguage;
```

Constructor *DemandSpace()*

This method is the constructor of the class. It runs the Object Database and assigns its reference to the *objectDatabase*. In addition, it gets a reference to the database query language and assigns it to the *objectQueryLanguage*. If one of the two references is null, the constructor raises an exception, i.e. the constructor succeeds if and only if there are established Object Database and its Object Query Language.

Pseudo code

```
public DemandSpace ( )
throws DemandDispatcherException
{
 Run the Object Database and assign its
 reference to the objectDatabase.
 If objectDatabase = null then
   throws DemandDispatcherException.
 If exists objectDatabase.QueryLanguage then
 objectQueryLanguage=
   objectDatabase.QueryLanguage.
}
```

Class *DispatcherEntry*

The class *DispatcherEntry* is the design solution for unifying the different kinds of demands as one entry. The class is derived from the *Adapter* pattern [19, 20] (see Fig.3.12). It provides a mechanism for wrapping the demands in an appropriate format for storing them in the DS. The *DispatcherEntry* class implements the *Serializable* interface, which makes the objects, instantiated from the class, persistent and appropriate for storing in the DS. Due to the nature of DS as an Object Database and the requirements for persistency, the class *DispatcherEntry* must fulfill the following serialization requirements:

- It must have a default no argument constructor.
- All the instance variables – class fields, must be public.
- All the instance variables must be serializable, i.e. a DispatcherEntry object cannot have primitive variables.

In our design, the DispatcherEntry class holds one and only one demand, this being *pending*, *in process* or *computed.* It holds it as a *serializable* object, i.e. an object that could be saved permanently. Therefore, we address two major concerns here:

- First, we assure that the demands and results will be permanently stored in the Demand Space until they are required. Therefore, they will be not lost in case of a distributed node failure or restart.
- Second, we unify both the demands and results as one entity.

Fig.3.12 depicts *wrap* associations between the *DispatcherEntry* class and *Demand* and *Result* classes. The Demand class represents the *pending* and *in process* demands, and the Result class represents the *computed* demands. Both classes implement the *Serializable* interface. Except a wrapped demand, the class *DispatcherEntry* also holds additional vital information like an entry's GUID number and a flag for distinguishing the demands by their states - *pending*, *in process* or *computed*. The GUID number and flag are used by the TAs for locating the appropriate demand and for determining its state (see the *DemandState* class).

The class *DispatcherEntry* implements the following methods and data fields:

Data field *oUniqueID*

This data field holds the Global Unique Identifier (GUID) number assigned to the entry object (respectively to the demand wrapped by this entry). This GUID is the key used for uniquely identifying the demand.

Pseudo code

```
public Uuid oUniqueID;
```

Data field *dState*

This data field holds the demand state – *pending*, *in process* or *computed*.

Pseudo code

```
public DemandState dState;
```

Data field *oDemand*

This data field holds the demand object.

Pseudo code

```
public Serializable oDemand;
```

Constructor *DispatcherEntry()*

This method is the constructor of the class. Its primary purpose is the initialization of the class' data fields.

Pseudo code

```
public DispatcherEntry (
  Uuid theID, Serializable theDemand,
  DemandState newState)
{ oUniqueID = theID.
  oDemand = theDemand..
  dState = newState.
}
```

Class *DemandState*

The class *DemandState* is our design solution for enumerating the demand states. This class defines the three states - *pending*, *in process* and *computed*, as public instances of the same class. In addition, the class provides functionality for determining the current state. The constructor of the class is designed as private, thus preventing from creation of other states, i.e. we enforce the use of states being already created and restrict the creation of new ones.

The class *DemandState* implements the following methods and data fields:

Data field *sState*

This data field keeps the current state.

Pseudo code

```
private String sState;
```

Data field *PENDING*

This data field holds an instance of the class representing the *pending* state.

Pseudo code

```
private static final String
   STR_PENDING = "pending";
public static final DemandState PENDING
   = new DemandState (STR_PENDING);
```

Data field *INPROCESS*

This data field holds an instance of the class representing the *in process* state.

Pseudo code

```
private static final String
    STR_INPROCESS = "inprocess";
public static final DemandState INPROCESS
   = new DemandState (STR_INPROCESS);
```

Data field *COMPUTED*

This data field holds an instance of the class representing the *computed* state.

Pseudo code

```
private static final String
    STR_COMPUTED = "computed";
public static final DemandState COMPUTED
   = new DemandState (STR_COMPUTED);
```

Constructor *DemandState()*

This method is the constructor of the class. It accepts a new state as a parameter and assigns it to the *sState* private data field. The constructor is designed as private.

Pseudo code

```
private DemandState (String newState)
{
 Assign newState to sState.;
}
```

Methods for determining the state

The class implements three boolean methods for determining the current state - *isPending*, *isInProcess* and *isComputed*.

Pseudo code

```
public boolean isPending();
public boolean isInProcess();
public boolean isComputed();
```

Exceptions

In our design solution for the Demand Dispatcher package, we defined two exceptions – *DemandDispatcherException* and *DSException*. These exceptions inherit the *DMSException* (see Fig.3.12). The *DemandDispatcherExcaption* can be thrown by the methods of the *DemandDispatcher* class. The *DSException* can be thrown by the methods of the *DemandDispatcher* and *DemandSpace* classes.

3.3.4. JINI Transport Agent

One of our TA design solutions is based on JINI [3, 7, 34]. The JINI Transport Agent (JTA) has all the characteristics of a JINI service. In addition, it is able to serve the DD, DGs and workers as a messenger [2].

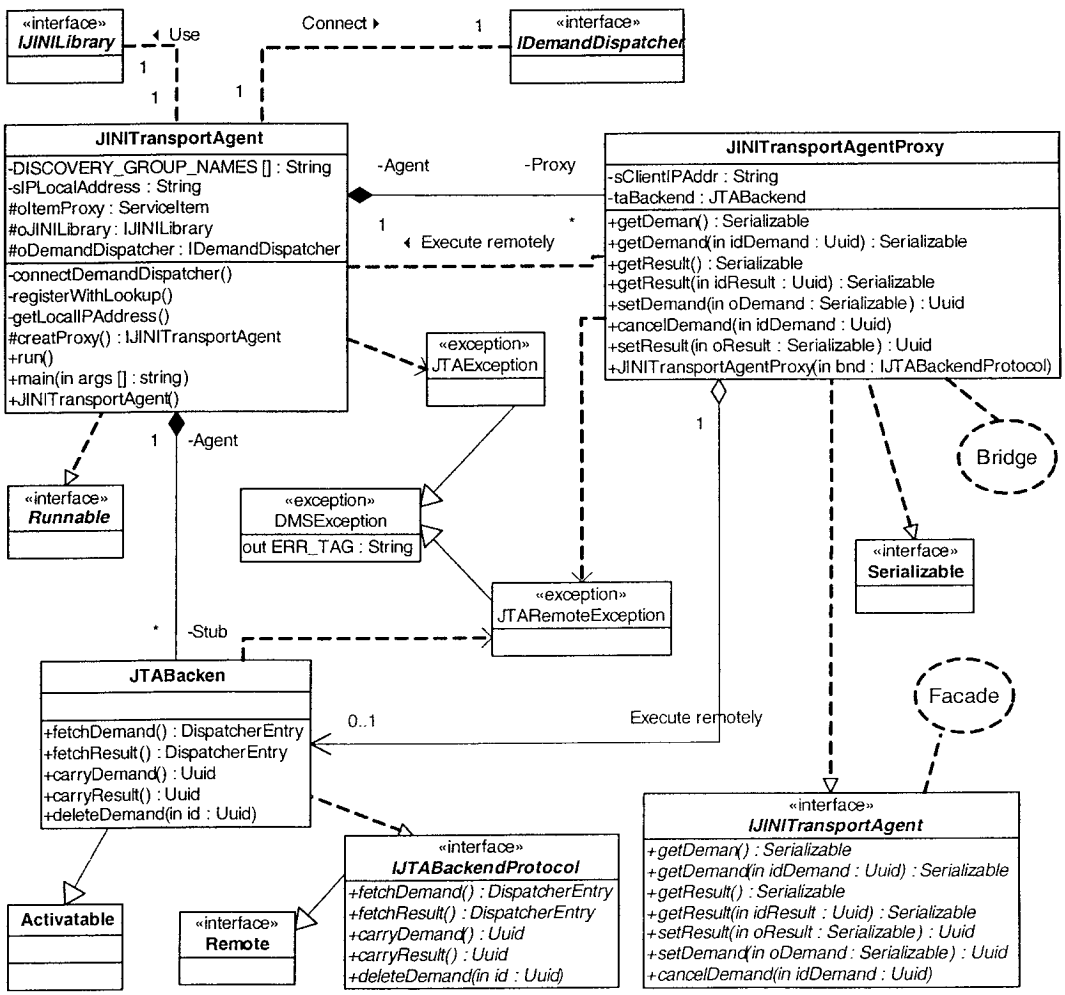


Fig.3.13. JINI Transport Agent Class Diagram

The JTA implements the main characteristics applicable to the DMS' TAs. Hence, it is a stand-alone component that exposes a common interface to the DD, DG and workers for demand migration. The UML class diagram [19, 40] above depicts the JINI Transport Agent package's classes and their relationships.

3.3.4.1. Design Rationale

The following is a list of questions that raised during the design process of the JINI Transport Agent package and their reasonable answers implemented in our design.

Why JINI? JINI is a distributed computing technology that forms an open architecture for federating services in a distributed system (see section 4.1.1). JINI is a pure Java technology and integrates easily with the GIPSY, which is entirely implemented in Java. Hence, we decided to design our primary TA solution based on JINI due to its Java nature, .i.e. we did not have to design additional Java stubs or use IDL interfaces that are necessary for other distributed technologies.

The JINI package. The JTA should rely on the JINI package for all the JINI related functions, i.e. we should not integrate the JINI functions in the JTA. This is important because JINI evolves and changes. Hence, in our JTA design, in order to prevent a JTA modification due to a JINI upgrade, we should heavily rely on the JINI Package. The last will tackle with future changes in the JINI environment.

JINI Transport Agent is a JINI service. A JINI service is implementation of the most important concept within the JINI architecture (see section 4.1.1.1). This concept defines the JINI service as the provider of distributed computing. Therefore, the JINI Transport Agent (JTA) has all the characteristics of a JINI service [3, 7]. JINI services are defined via an interface, and the implementation of a proxy adhering to that interface. Hence, our JTA should expose its functionality through an interface and should rely on a proxy that fulfills presentation functions. This necessitates the use of the *Facade* and *Bridge* pattern [19, 20] (see Fig.3.13). The proxy is downloaded by the client – a DG or worker. Further, the proxy communicates with its mentor – the JTA.

GIPSY lookup service. In order to be available, the JINI services register on lookup services (see section 4.1.1). Hence, we should register our JTA in a lookups service (LUS) as well. This LUS must be used only by the GIPSY' JTAs, hence we should run one GIPSY lookup service for all the JTAs within the GIPSY.

The JTA interface. The JTA as a regular TA should expose its TA functionality via its interface. The JTA will work with those workers and DGs that adhere to the JTA interface.

The role of RMI. As we said in the section above, JINI evolves and today it relies on a number of mechanisms for distributed processing. One of these is *Remote Method Invocation* (RMI), which is supported by all the JINI releases. Therefore, in our design we should rely on RMI for having full JINI compatibility. RMI ensures that two processes running on separate machines can exchange invocation requests and results (see section 4.1.1.2). The use of RMI requires a design of a special back-end class implementing the RMI interface called *Remote.* Hence, in our design we should implement a back-end class (see the *JTABackend* class in Fig.3.13).

Downloadable service proxy. JINI requires a JINI service to implement an arbitrary *Serializable* object called service proxy [3, 7]. Therefore, in our design we should have a design solution for a JTA proxy, since our JTA is a JINI service. JINI transports the JTA proxy from the JTA's machine to the GIPSY node's machine – a machine running a DG or worker. The DGs and workers use that proxy locally and transparently as they use the real JTA, i.e. they make method calls on the proxy. The proxy just transmits calls across the network to the real JTA. The real JTA and its proxy communicate internally in order to transport demands or results (see Fig.3.14).

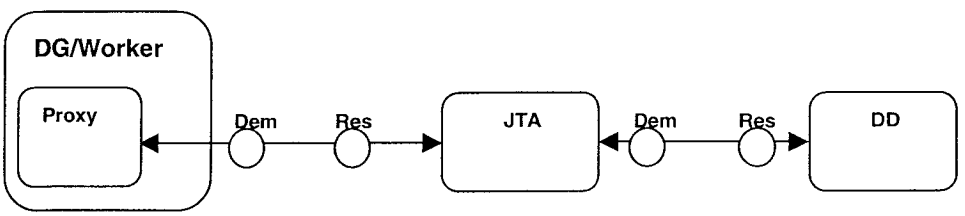


Fig.3.14. JTA – JTA Proxy Communication

Since the DGs and workers use the JTA via its proxy, the JTA proxy must implement the JTA interface, which exposes the JTA's functionality. Hence, the JTA's proxy must be designed as a *Serializable* class implementing the JTA interface.

The JINI class ServiceItem. The JINI architecture necessitates the use of this class as a container of the service proxy. An instance of this class is passed for registration to the JINI LUS [3, 7, 34]. Hence, in our design we should implement this class as a wrapper for the JTA proxy.

JTA security. The JINI Transport Agent is Java based. Therefore, its security mechanism should be tightly related to the Java Virtual Machine security mechanism. This mechanism consists of three key components such as Class Loader, Class File Verifier and Security Manager [21, 22, 3, 7]. From the JINI programmer perspective the most important is the Security Manager. The Security Manager is a run-time manager that applies the restrictions based on security policy statements. It supports a *policy-driven* security model based on special policy files that being associated with the process to be run at startup. The Security Manager prevents operations by throwing an exception. It simply returns if the operation is permitted, but throws an exception if the operation is not permitted. JINI downloads from a remote location only if the Security Manager has been set. Therefore, the JTA must sets up the Security Manager during its construction or before the first execution of any JTA method. In addition, the JTA should rely on a policy file for providing the security policy to its Security Manager. Every JTA runs in conjunction with such a policy file.

3.2.4.2. Detailed Design

Before going over the detailed design, there are some relationships, depicted by the UML class diagram (see Fig.3.13), that require a detailed explanation. The UML diagram depicts two *execute-remotely* relationships, those demonstrating the remote collaboration between the proxy and the main class. Those relationships are in consequence of our decision to make the JTA fully JINI compatible. The first relationship is between the classes *JINITransportAgentProxy* and *JINITransportAgent*. It is an association (see the dashed line in Fig.3.13) that depicts an indirect remote collaboration between the proxy and its mentor. The second relationship is between the classes *JINITransportAgentProxy* and *JTABackend*. This

relationship depicts a remote collaboration based on RMI, which enforced the design of the *JTABackend* class as an RMI back-end class. The last implements the *Remote* interface, since the methods of this class will be remotely callable. The class *JTABackend* is an inner class for the class *JiniTransportAgent*. It is used by the proxy for calling its mentor (the JTA) remotely.

The following elements describe the JINI Transport Agent package's classes and their attributes and methods.

Interface *IJINITransportAgent*

The *IJINITransportAgent* interface is the package's entry interface. This interface is the highest JTA abstraction level. Most of the methods exposed by the interface are focused on demand migration, i.e. get demands from and set demands in the Demand Space (DS). The *DGs and workers use the IJINITransportAgent interface* first to find the JTA within the JINI federation of services, and second to call the JTA functions. The JTA hides the complex remote communication between the JTA and its proxy, i.e. the DGs and workers simply use the exposed by the interface functions as their own. The interface *IJINITransportAgent* exposes the package's functionality, by relaying on the methods, listed below.

Method *getDemand()*

This method gets a *pending* demand from the DS. There are two overloading *getDemand* methods in the interface. Whereas the first one does not accept any parameters, the second one accepts the GUID number of the desired demand. Hence, the first method gets any *pending* demand and the second gets a well-specified one. The method returns the found demand in a *Serializable* format.

Pseudo code

```
public Serializable getDemand();

public Serializable getDemand(
  Uuid idDemand);
```

Method *getResult()*

This method gets a *computed* demand from the DS. Similarly, to the previous case, there are two overloading *getResult* methods in

Pseudo code

```
public Serializable getResult();

public Serializable getResult()
  Uuid idDemand);
```

the interface - first one accepting no parameters and the second one accepting the GUID number of the desired *computed* demand as a parameter. Hence, the first method gets any *computed* demand and the second gets a well-specified one.

Method *setDemand()*

This method sets a *pending* demand in the DS. The demand to be set is passed as a *Serializable* parameter and the method returns as result its GUID. The last is generated by the Dispatcher Proxy (see section 3.3.3).

Pseudo code

```
public Uuid setDemand( Serializable oDemand);
```

Method *setResult()*

This method sets a *computed* demand in the DS. The method accepts two parameters – the demand to be set passed as a *Serializable* parameter and its GUID. The GUID is the one of the *pending* demand used for generating the result.

Pseudo code

```
public Uuid setResult(
  Serializable oResult, Uuid idResult);
```

Method *cancelDemand()*

This method deletes permanently any kind of demand from the DS. The method accepts one parameter – the demand's GUID, specifying the demand to be deleted.

Pseudo code

```
public void cancelDemand(Uuid idDemand);
```

Method *setClientIPAddress()*

This method sends the client's IP address – the one of a worker or DG, to the JTA. The JTA uses internally this address for a notification purpose.

Pseudo code

```
public void setClientIPAddress(
  String sIPAddress);
```

Class *JiniTransportAgent*

The class *JINITransportAgent* is the main class in the package. It instantiates a standalone object that is the real JTA. It contains the java *main()* function. This class implements the functionality of a TA (see section 2.2.3) and the one of a JINI service. As a JINI service, it will find the lookup service and publish its proxy. The class wraps the proxy (see the *JINITransportAgentProxy* class), which is used to implement the interface *IJINITransportAgent* describing the public JTA functionality. In addition, the class *JINITransportAgent* establishes the connection with the DD. The class implements the *Runnable* Java interface (see Fig.3.13). The purpose is to keep the JTA running, even when it is in an idle mode. In the *main()* function we instantiate a *thread* [23, 41] targeted to an instance of the class *JINITransportAgent*, i.e. we ensure that the JTA will not simply stop when the *main()* method finishes. Therefore, in our design the JTA implements the *Runnable* interface and causes a simple thread to be started that sleeps forever, but keeps the JTA alive.

For all the JINI-oriented operations the class relies on the JINI Library (see section 3.3.2). The class *JINITransportAgent* implements the following methods and data fields:

Data field *oDemandDispatcher*

This data field holds a reference to implementation of the *IDemandDispatcher* interface, i.e. it holds a reference to the DD (see section 3.3.3). This reference is used for accessing all the DD's functions.

Pseudo code

```
protected static IDemandDispatcher
oDemandDispatcher;
```

Data field *oJINILibrary*

This data field holds a reference to the implementation of the *IJINILibary* interface, i.e. it holds a reference to the JINI Library (see section 3.3.2).

Pseudo code

```
protected static IJINILibrary oJINILibrary;
```

Data field *oItemProxy*

This data field holds an instance of the JINI *ServiceItem* class [3, 7, 34]. This instance wraps the proxy. It is passed to the lookup

Pseudo code

```
protected ServiceItem oItemProxy;
```

service during the registration process.

Method *main()*

This method implements the *main()* Java function. The method creates a new instance of the *JINITransportAgent* class and starts a background thread to keep the JTA alive.

Pseudo code

```
public static void main(String args[]) {
1. Create a new instance of
   JINITransportAgent.
2. Create a new thread targeted to the
   JINITransportAgent instance.
3. Start the thread.
}
```

Method *run()*

This method is the *run()* method inherited from the *Runnable* interface. The method simply starts the thread in an idle mode – make it sleeping forever.

Pseudo code

```
public void run()
{
  Make the thread sleeping forever.
}
```

Method *connectDemandDispatcher()*

This method connects the JTA with the Demand Dispatcher by assigning the DD reference to the *oDemandDispatcher* data field.

Pseudo code

```
public void connectDemandDispatcher () {
 1. Find the DD.
 2. Get a reference to it.
 3. Assigns that reference to
    oDemandDispacther data field.
}
```

Method *registerWithLookup()*

This method registers the JTA on the GIPSY lookup service (see section 3.3.4.1) by using the *oJINILibrary* data field.

Pseudo code

```
protected synchronized void
  registerWithLookup() {
Use oJINILibrary to register the JTA on the
GIPSY LUS. }
```

Method *createProxy()*

This method constructs the proxy object and returns it as a result. For the creation of the proxy, the method should create an instance of the back-end class *JTABackend* and pass it to the proxy's constructor, so the proxy can call back to it.

Pseudo code

```
protected IJINITransportAgent createProxy() {
1. Create an instance of JTABackend.
2. Pass the JTABackend instance to the
   JiniTransportAgentProxy constructor, and
   create a JTA proxy.
3. Return the created proxy as a result.
}
```

Constructor *JiniTransportAgent()*

This method is the constructor of the class. It sets the security manager, connects with the DD and registers the proxy on the GIPSY lookup service (see section 4.2.3).

Pseudo code

```
public JINITransportAgent ()
throws JTAException {
1. Set the security manager.
2. Call connectDemandDispatcher().
3. Call registerWithLookup()
}
```

Interface *IJTABackendProtocol*

This intreface defines the remote communications protocol between the client-side stub and the service-side object, i.e. it defines the protocol that the proxy object will use to communicate with the back-end remote object. The interface extends the RMI *Remote* interface [21, 23]. All the methods exposed by the interface can be called remotely. Therefore, a proxy with a reference to the *IJTABackendProtocol* interface can invoke all the methods exposed by the interface regardless of where the *IJTABackendProtocol* implementation physically resides. All the interface's methods throw a *JTARemoteException* exception.

Method *fetchDemand()*

This method fetches a *pending* demand from the DS and carries the demand from the JTA to its proxy.

Pseudo code

```
public DispatcherEntry fetchDemand(
    Uuid idDemand, String sSenderIP)
throws JTARemoteException;
```

Method *fetchResult()*

This method fetches a *computed* demand from the DS and carries the demand from the JTA to its proxy.

Pseudo code

```
public DispatcherEntry fetchResult(
    Uuid idResult, String sSenderIP)
throws JTARemoteException;
```

Method *carryDemand()*

This method carries a *pending* demand from the JTA proxy to the JTA and stores it in the DS.

Pseudo code

```
public Uuid carryDemand(
  IWorkDemand oDemand, String sSenderIP)
throws JTARemoteException;
```

Method *carryResult()*

This method carries a *computed* demand from the JTA proxy to the JTA and stores it in the DS.

Pseudo code

```
public Uuid carryResult(
    IWorkResult oResult, Uuid idResult,
    String sSenderIP)
throws JTARemoteException;
```

Method *deleteDemand()*

This method deletes permanently a demand from the DS. The demand to be deleted is specified by its GUID.

Pseudo code

```
public void deleteDemand (Uuid id)
throws JTARemoteException;
```

Class *JTABackend*

This class implements the *IJTABackendProtocol* interface. This is the class, which is used by RMI to ensure a *service-side* execution, by generating stubs, which are

transported to the client – a DG or worker. An instance of this class is used by the JTA proxy to perform remote executions on the JTA functions, i.e. the following methods implemented by the class *JTABackend* run on the JTA side. This is possible due to the fact that the *JTABackend* class inherits the RMI's class *Activatable*, which makes the *JTABackend's* methods callable from remote Java Virtual Machines [21, 22, 23]. The class *JTABackend* is designed as an inner class, i.e. it has full access to all the attributes and methods of its outer class – *JINITransportAgent*.

Method *fetchDemand()*

This method implements the *fetchDemand* method of the *IJTABackendProtocol* interface (see the *IJTABackendProtocol* interface).

Pseudo code

```
public DispatcherEntry fetchDemand(
    Uuid idDemand, String sSenderIP)
throws JTARemoteException {
  Use the oDemandDispatcher data field
  from JINITransportAgent class to execute
  the DemandDiaptcher's method
  readDemand.
}
```

Method *fetchResult()*

This method implements the *fetchResult* method of the *IJTABackendProtocol* interface (see the *IJTABackendProtocol* interface).

Pseudo code

```
public DispatcherEntry fetchResult(
    Uuid idResult, String sSenderIP)
throws JTARemoteException {
  Use the oDemandDispatcher data field
  from JINITransportAgent class to execute
  the DemandDiaptcher's method
  readResult (idResult).}
```

Method *carryDemand()*

This method implements the *carryDemand* method of the *IJTABackendProtocol* interface (see the *IJTABackendProtocol* interface).

Pseudo code

```
public Uuid carryDemand(
 IWorkDemand oDemand, String sSenderIP)
throws JTARemoteException {
  Use the oDemandDispatcher data field
  from JINITransportAgent class to execute
  the DemandDiaptcher's method
  writeDemand (oDemand).}
```

Method *carryResult()*

This method implements the *carryResult* method of the *IJTABackendProtocol* interface (see the *IJTABackendProtocol* interface).

Pseudo code

```
public Uuid carryResult( IWorkResult oResult,
  Uuid idResult,  String sSenderIP)
throws JTARemoteException {
  Use the oDemandDispatcher data field
  from JINITransportAgent class to execute
  the DemandDiaptcher's method
  writeResult (idResult, oResult).}
```

Method *deleteDemand()*

This method implements the *deleteDemand* method of the *IJTABackendProtocol* interface (see the *IJTABackendProtocol* interface).

Pseudo code

```
public void deleteDemand (Uuid id)
throws JTARemoteException {
  Use the oDemandDispatcher data field
  from JINITransportAgent class to execute
  the DemandDiaptcher's method
  cancelDeman (id).}
```

Class *JiniTransportAgentProxy*

This class is our design solution for the JTA proxy. It implements two interfaces – *IJINITransportAgent* and *Serializable*. The *Serializable* interface assures that an instance of the class could be sent to each DG or worker attempting to connect to the JTA. The class has a public no-argument constructor, due to its *Serializable* nature [3, 7, 23]. The *IJINITransportAgent* interface is the JTA interface known by the DGs and workers. In our design, the class *JINITransportAgentProxy* is designed as a static, no public and inner class for the *JINITransportAgent* class. The argument used here is fine – the DGs and workers gain access to an instance of this proxy at run-time via serialization and code downloading [3, 7], and the inner class has full access to all the attributes and methods of its outer class – *JINITransportAgent*. The class *JINITransportAgentProxy* is designed as a static to demonstrate that the class is nested simply for structured convenience – not for any run-time associations between the nested and outer class, which simply wraps and publishes it.

The class *JINITransportAgentProxy* implements the following methods and data fields:

Data field *sClientIPAddress*

This data field holds the IP address of the machine running the client - a DG or worker, where the proxy is moved.

Pseudo code

```
private String sClientIPAddress;
```

Data field *taBackend*

This data field holds a reference to a backend object implementing the *IJTABackendProtocol* interface.

Pseudo code

```
private IJTABackendProtocol taBackend;
```

Constructor *JINITransportAgentProxy()*

There are two constructors implemented by the class. The first one is by default with no parameters and is used for the serialization of the proxy. The second one accepts one parameter that is the backend object created by the JTA.

Pseudo code

```
public JINITransportAgentProxy()
{
}

public JINITransportAgentProxy(
  IJTABackendProtocol backend)
{
  Assign the backend to taBackend.
}
```

Method *getDemand()*

This method implements the *getDemand* methods of the *IJINITransportAgent* interface (see the *IJINITransportAgent* interface).

Pseudo code

```
public Serializable getDemand() {
 1. Call taBackend.fetchDemand( null,
     sClientIPAddr).
 2. Return the fetched demand if any.}

public Serializable getDemand( Uuid id) {
 1. Call taBackend.fetchDemand( id,
     sClientIPAddr).
 2. Return the fetched demand if any.}
```

Method *getResult()*

This method implements the *getResult* methods of the *IJINITransportAgent* interface (see the *IJINITransportAgent* interface).

Pseudo code

```
public Serializable getResult() {
 1. Call taBackend.fetchResult( null,
     sClientIPAddr).
 2. Return the fetched result if any.}
public Serializable getResult( Uuid id) {
 1. Call taBackend.fetchResult( id, sClientIPAddr).
 2. Return the fetched result if any.}
```

Method *setDemand()*

This method implements the *setDemand* method of the *IJINITransportAgent* interface (see the *IJINITransportAgent* interface).

Pseudo code

```
public Uuid setDemand(
  Serializable oDemand)
{
 Call taBackend.carryDemand (oDemand,
 sClientIPAddr).
}
```

Method *setResult()*

This method implements the *setResult* method of the *IJINITransportAgent* interface (see the *IJINITransportAgent* interface).

Pseudo code

```
public Uuid setResult(
  Serializable oResult, Uuid idResult)
{
 Call taBackend.carryResult (oResult,
 idResult, sClientIPAddr).
}
```

Method *cancelDemand()*

This method implements the *cancelDemand* method of the *IJINITransportAgent* interface (see the *IJINITransportAgent* interface).

Pseudo code

```
public void cancelDemand(Uuid idDemand)
{
 Call taBackend.cancelDemand (idDemand).
}
```

Method *setClientIPAddress()*

This method implements the *setClientIPAddress* method of the *IJINITransportAgent* interface (see the *IJINITransportAgent* interface).

Pseudo code

```
public void setClientIPAddress(
  String sIPAddress)
{
 Assign the sIPAddressto to sClientIPAddress.
}
```

Exceptions

In our design solution for the JINI Transport Agent package, we designed two exceptions – *JTAException* and *JTARemoteException*. These exceptions inherit the *DMSException* (see Fig.3.13). The *JTAException* can be thrown by the methods of the *JINITransportAgent* class. The *JTARemoteException* can be thrown by the methods of the *JINITransportAgentProxy* and *JTABackend* classes.

3.4. Process View

The process architectural view [18] consists of the processes and threads that form the system's concurrency and synchronization mechanisms, as well as their interactions. In this architectural view, we address issues such as concurrency and parallelism, DMS' components dependency, scalability, consistency and fault-tolerance (e.g. isolation of functions and faults, reliability with accuracy). In our process architectural view the independent flows of control such as *threads* and *processes* are modeled as *active objects*. An *active object* is an instance of an *active class* [40] (see Fig.3.15 and Fig.3.16). We propose a multi-process architecture, where the processes are independent standalone executables. In addition, they are "volunteers" that could register to and unregister from the DMS at any time.

In the course of the design process, we designed two process views. The first one is more general and it corresponds to our general design concept. The second one corresponds to our detail design and it is so-called JTA process view, due to the fact we stuck with the detail design of the JINI Transport Agent (see section 3.3.4). The process view diagrams depict two levels of abstractions (see Fig.3.15 and Fig.3.16):

- The first level, called "GIPSY execution nodes", represents the DGs and workers as processes relying on the DMS for dispatching and receiving demands and results (*computed* demands).
- The second level, called "DMS middleware", is the run-time state of our DMS in terms of processes and threads.

3.4.1. Design Rationale

The following is a list of questions that raised during the design of the DMS process view and their reasonable answers implemented in our design.

Processes. Processes are heavyweight system run-time entities [41]. In our DMS, a process runs on the Java machine and requires time, memory, and CPU resources. The processes in our DMS communicate by relying on mechanisms provided by the host Java machine and distribution middleware [24]. This communication is called *interprocess communication*. All the *interprocess communications* within the DMS must be *asynchronous* and the processes should not use shared resources, except those exposed by the Demand Space. Therefore, the *interprocess communication* [41] will be possible in a high concurrent manner, since the processes do not have to synchronize their state, and a process cannot corrupt the state of another.

Threads. The threads are spawned lightweight executing processes that operate within the memory space of the running process [41]. Some of the DMS processes in order to improve its ability to perform parallel tasks should spawn threads [24]. Those threads should communicate asynchronously with the process spawning them (host process) and possibly with other processes and threads. Therefore, the DMS processes should spawn threads when they need to perform parallel processing.

DMS components run as processes. The DMS architecture necessitates *hot-plugging* (see section 2.1 and section 2.3). Therefore, all the components that implement *hot-plugging* should be "volunteers". This is possible only with stand alone independent components, i.e. components that control their lifetime. Therefore, all the DMS components implementing *hot-plugging* must be designed as processes. Hence, the processes run within the DMS should be all the TAs and DS. In addition, all the distribution technologies should establish a middleware that run in a background mode, i.e. it is a process as well.

The DP is a thread not a process – why? The Dispatcher Proxy (DP) makes the use of DS transparent (see section 2.3). It hides the details of the *interprocess communication* between the DS clients and DS itself, where the DS clients are TAs, DGs and workers. Therefore, the DS clients should communicate with the DP locally by using the DP functions as their own. In addition, the DS clients should be not blocked by the execution of any DP function. Hence, the DP should run as a thread spawned by a DS client.

DS parallelism and concurrency. Concurrent access is vital for our Demand Dispatcher (DD). In our design we assure concurrent access to the DS, by granting each DS client with a single DP. Each DP runs as a thread. The DP is alive untill the TA is connected to the DMS. In addition, the DS must implement internal parallelism allowing simultaneous serving of many clients.

3.4.2. General Process View

The processes and threads depicted by Fig.3.15 are the architectural components that can be uniquely addressed as architectural design elements. Those processes and threads communicate through a set of well-defined communication mechanisms: synchronous and asynchronous message-based communication services, remote procedure calls and event broadcasts. This process view does not make any assumptions neither about the processes collocation in the same processing node, neither about the threads collocation in the same process.

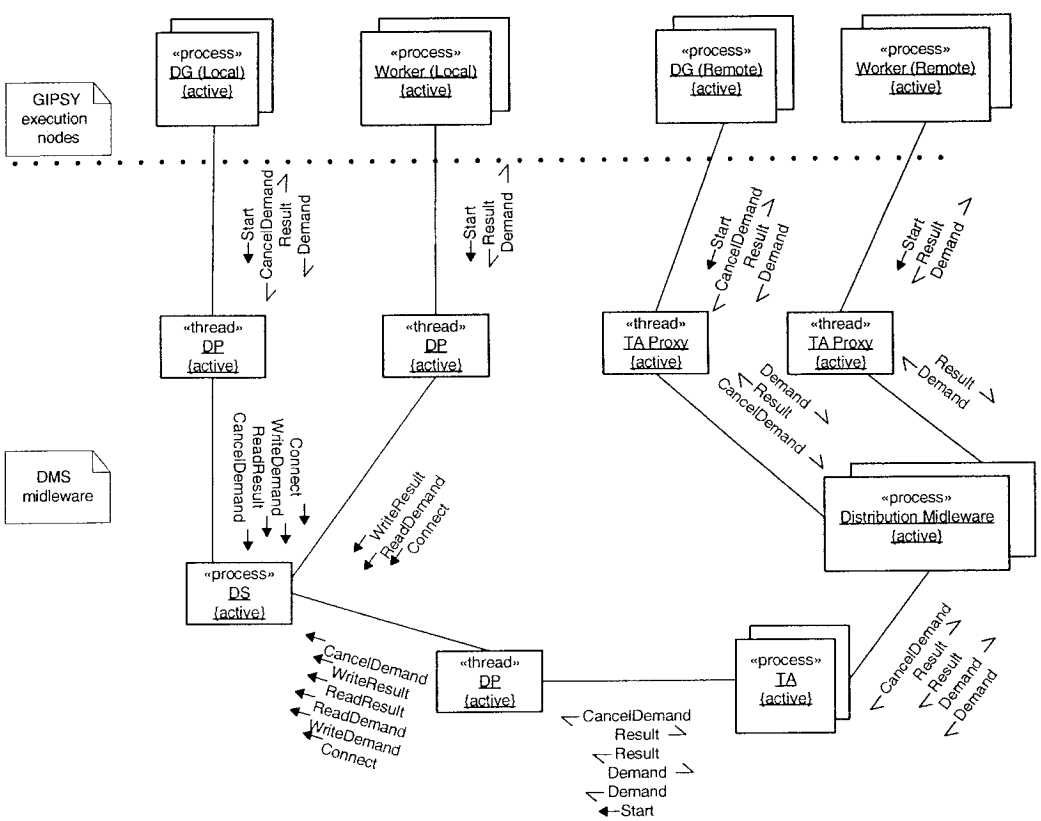


Fig.3.15. General Process View Diagram

Process DS. This process is the runtime presentation of the Demand Space. The DMS runs the DS as a *Singleton* [19, 20] (see section 3.3.3.1), i.e. one process serves all the DS clients.

Thread DP. This thread is the runtime presentation of the DP. It is spawned by a TA, DG or worker. There is one instance of this thread per any client connected to the DS. The DP thread communicates with the DS in a *synchronous manner* by procedure calls – remote or local.

Process TA. This process is the runtime presentation of the TA. The DMS runs multiple TAs, each one related to a distribution middleware. The communication between a middleware and TA is based on *asynchronous messages*, event broadcasts and remote procedure calls. The TA process spawns a DP thread, which is used to retrieve and set asynchronously demands and results. A TA spawns one single DP thread.

Thread TA Proxy. This thread is the TA stub exposing the TAs functionality to the DGs and workers in a transparent manner, i.e. it hides the remote collaboration with the real TA (see section 3.3.1). The TA Proxy thread is spawned by the DGs and workers in their address space. There is only one TA Proxy spawned by a DG or worker, i.e. each TA Proxy is associated with a DG or worker. The TA Proxy communicates with the real TA via the distribution middleware. All the communications performed by this thread are *asynchronous*.

Process Distribution Middleware. This process is the runtime presentation of the distributed technologies. Each distributed technology establishes middleware between the remote machines, i.e. runs a middleware process. The DMS components interact with those processes *asynchronously*.

Processes DG and Worker. These processes are the run time presentation of the GIPSY execution nodes – DGs and workers.

3.4.3. JTA Process View

The JTA process view is the run time picture of our DMS detailed design. Fig.3.16 depicts processes and threads that can be uniquely addressed as detailed design elements (see section 3.3).

Process DemandSpace. This process is the run-time instance of the *DemandSpace* class (see section 3.3.3). It is similar to the DS process from the general process view.

Thread DispatcherProxy. This thread is the run-time instance of the *JINITransportAgentProxy* class (see section 3.3.3). It is similar to the DP thread from the general process view.

Process JINI. This process is the runtime presentation of the JINI distributed technology. It is similar to the distribution middleware process from the general process view.

Thread JINITransportAgentProxy. This thread is the run-time instance of the *JINITransportAgentProxy* class (see section 3.3.4). It is similar to the DP Proxy thread from the general process view.

Process JINITransportAgent. This process is the run-time instance of the *JINITransportAgent* class (see section 3.3.4). It spawns multiple *JTABackend* threads and one single *DispatcherProxy* thread. The *JINITrnasportAgent* process controls only the lifetime of those threads, which do the actual TA work. In addition, it associates each *JTABackend* thread with a *JINITransportAgentProxy* thread.

Thread JTABackend. This thread is the run-time instance of the *JTABackend* class (see section 3.3.4). The *JINITransportAgent* process spawns such a thread for every *JINITransportAgentProxy* thread. The last communicates with its corresponding *JTABackend* thread via the JINI distribution middleware (see Fig.3.16). In addition, the *JTABackend* thread communicates with the *DispatcherProxy* thread for retrieving

and setting demands and results. All the communications performed by the thread of *JTABackend* are *asynchronous.*

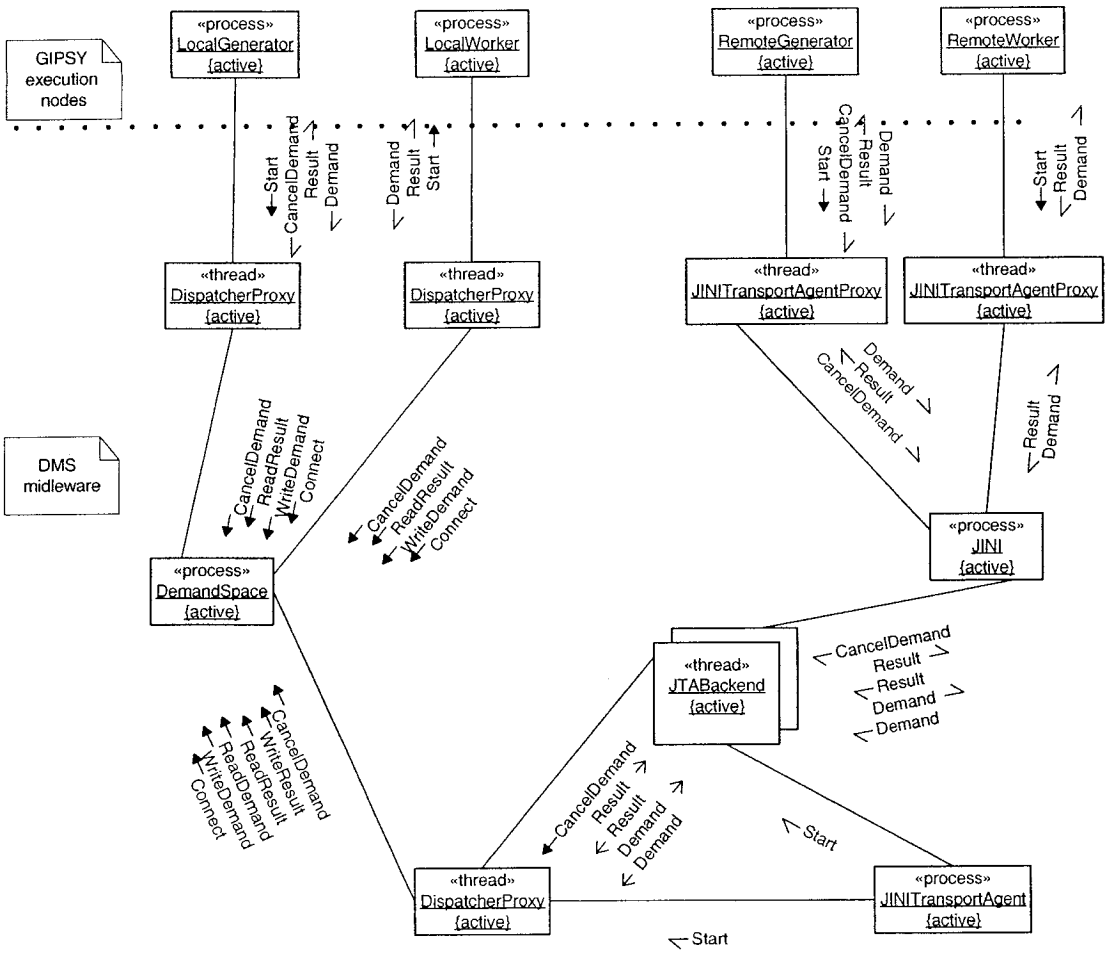


Fig.3.16. JTA Process View Diagram

3.4.4. Reliability and Accuracy

Migrating demands in a heterogeneous environment requires very high reliability and accuracy. Our system will be reliable and accurate if it is able to deliver every demand to its destination, i.e. no demand could be lost due to any kind of fault or crash. We designed a DMS system consisting of loosely coupled components. Most of these components run as stand-alone components, depicted as processes in Fig.3.15 and Fig.3.16. The other DMS components are threads run by those processes. The processes interact with each other via *asynchronous* messages (see Fig.3.15 and Fig.3.16). The argument here is fine, since our goal was to achieve

minimum coupling. The correction of faults in such a system could be too expensive, risky or inconvenient. Therefore, we designed our loosely coupled architecture with a *fault-tolerance* error strategy. Our DMS minimizes the damages on any fault or crash by simply redoing the interrupted migration. Any component caused a failure, could be simply restarted, or its work could be directed to another component. For example if a TA crashed, the DD could re-dispatch the demand through another TA. It is possible because the DMS keeps the original demand permanently saved in the DS until its associated result is not received (see section 2.2.2).

3.4.5. Concurrency and Consistency

Since our DMS is based on distributed technologies, it is inherently concurrent [17, 21]. All the TAs exist independently and operate concurrently (see Fig.3.15). The TAs work simultaneously on the DS. In addition, all the TA Proxies work concurrently with the common TA. Another concurrency issue is coming from the distribution middleware processes, which are isolated from each other but work for migrating demands in a concurrent mode. Finally, the DS works simultaneously on multiple client requests. To achieve all these concurrency requirements we run all these components as processes or threads. These processes or threads communicate mainly in an *asynchronous manner* (see Fig.3.15 and Fig.3.16).

Every system has a consistency constraint that must be not violated. This constraint is related to the system *consistent state* [4]. The DMS will be in its *consistent state* if all the *in process* demands turn into *computed* ones, i.e. there is no *in process* demands that are on their way to being delivered. A consistency problem may arise when one of the components involved in the demand migration crashes or simply shuts down during a demand migration. Those components are the DP, TA, distribution middleware, PA proxy and worker (see Fig.3.15). In such a case, the DMS may simply re-dispatch the demand via another TA. This is possible due to the fact that the original demand is kept in the DS (see section 2.2.2).

3.4.6. Scalability

The DMS architecture will be scalable if it can accommodate any growth of future work increase. The distributed nature of our architecture assures high scalability. Our TAs are designed as processes (see Fig.3.15 and Fig.3.16) and we can easily

increase the workload by running new TAs. The only limitation is coming from the hardware resources and network communication. Network communication is often the key performance limitation (see section 5.3.2). All the network communication in our system is *asynchronous*, which increases the network performance, since there is no need of additional synchronous precautions. Although the networking performance of the system depends upon the physical architecture of the system, a big message should take longer to deliver than a short one. Another limitation is coming from the ability of the DS to handle increasing workload (see section 5.5.2). This problem could be easily solved if we run more DS spread among different machines.

3.4.7. Upgradeability

Upgradeability means that the system can be easily extended and modified. Upgradeability and distribution are related. Our DMS architecture assures high upgradeability. All the major DMS components run as independent processes (see Fig.3.15) communicating via well-defined interfaces (see section 3.3). Hence, we can easily integrate new components into the DMS and these components must adhere to the DMS interfaces. A possible upgrade could be adding new TAs to the system, or designing a new DS. The new TAs and DS must rely on the DP for their communication. In addition, we can easily extend the functionality of the already existing DMS components with the only restriction on preserving the old interfaces. All the DMS components, run as processes, implement *hot-plugging*. This increases the DMS architecture upgradeability, due to the fact we can integrate new components into the system without having to stop any DMS process.

3.4.8. Heterogeneity

Heterogeneity arises when a system should integrate heterogeneous components. Our DMS is heterogeneous due to the fact it relies on different distributed technologies for the implementation of TAs (see section 2.3). Fig.3.15 depicts that heterogeneity by associating each TA with a distribution middleware. The TAs rely on such distribution middleware for migrating demands.

Another heterogeneity issue comes from the distributed nature of our DMS. It is related to the ability to run DMS components on different platforms.

3.5. Deployment View

According to our design, the DMS runs on a network of computers called processing nodes. The "Deployment View" is a static deployment view of the run-time configuration of the GIPSY and DMS processing nodes and the components that run on those nodes. The following diagrams are UML deployment diagrams that depict an implementation-level of the DMS, i.e. they show the structure of the run-time system. The nodes in these diagrams are locations upon which components are deployed. The set of components that are allocated to a node as a group form a *distribution unit*. The diagrams in Fig.3.17 and Fig.3.18 depict the *distribution units* as machines, since each distribution unit is a single machine (computer) running the components of this unit. The components correspond to the processes and threads in the process view (see section 3.4).

From the conceptual view, we concluded two cases of the Demand Dispatcher (respectively Demand Space) distribution – local and remote (see section 2.3.1). Whereas Fig.3.17 depicts the "Remote DD" case, Fig.3.18 depicts the "Local DD" case.

3.5.1. Remote DD

There are three *distribution units* in the case "Remote DD" called respectively DG machine, DD machine and worker machine (see Fig.3.17). The machines are connected via a distribution protocol that is formed by one or more distributed technologies (see section 2.3.3). In this deployment view one DD machine serves many DG and worker machines.

The *distribution unit* of the DG machine runs a Demand Generator (DG), TA proxy and distribution middleware process. The distribution middleware process nests the TA proxy and makes the demand migration possible, i.e. the communication between the TA proxy and its mentor – the TA, run on a remote machine.

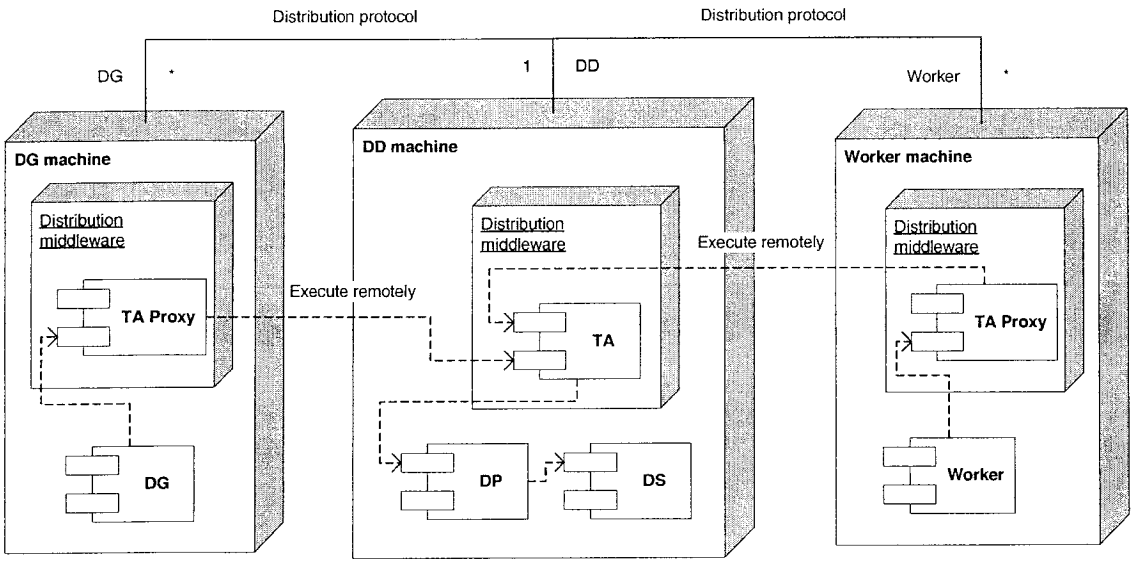


Fig.3.17. DMS Deployment Diagram "Remote DD"

The *distribution unit* of the DD machine forms the DMS middleware (see section 3.4). This machine runs the DS and TAs. Each TA is nested by a distribution middleware ensuring the remote communication between a TA and its proxy. In addition, there are DPs, each one being associated with a TA. Hence, this *distribution unit*, as a DD processing node of our DMS design, runs one DS component, many TA components - each one oriented to a distributed technology (see section 2.3.2), distributed middleware processes nesting a TA, and DP components connecting a TA with the DS.

The *distribution unit* of the worker machine is similar to the *distribution unit* of the DG machine. It runs a worker, TA proxy and distribution middleware process. The distribution middleware process nests the TA proxy.

3.5.2. Local DD

There are two *distribution units* in the "Local DD" case called respectively DG machine and worker machine (see Fig.3.18). The machines are connected via a distribution protocol that is formed by one or more distributed technologies (see section 2.3.3). In this deployment view, one DG machine works with many worker machines.

The *distribution unit* of the DG machine runs a Demand Generator (DG), TA and distribution middleware process. The distribution middleware process nests the TA. In addition, there are two DP components associated with the DG and TA and used for accessing the DS. The DG communicates with the DS via a DP. The TA is not needed here, since the DS is local and we do not have to cross any machine boundaries (see section 2.3.1).

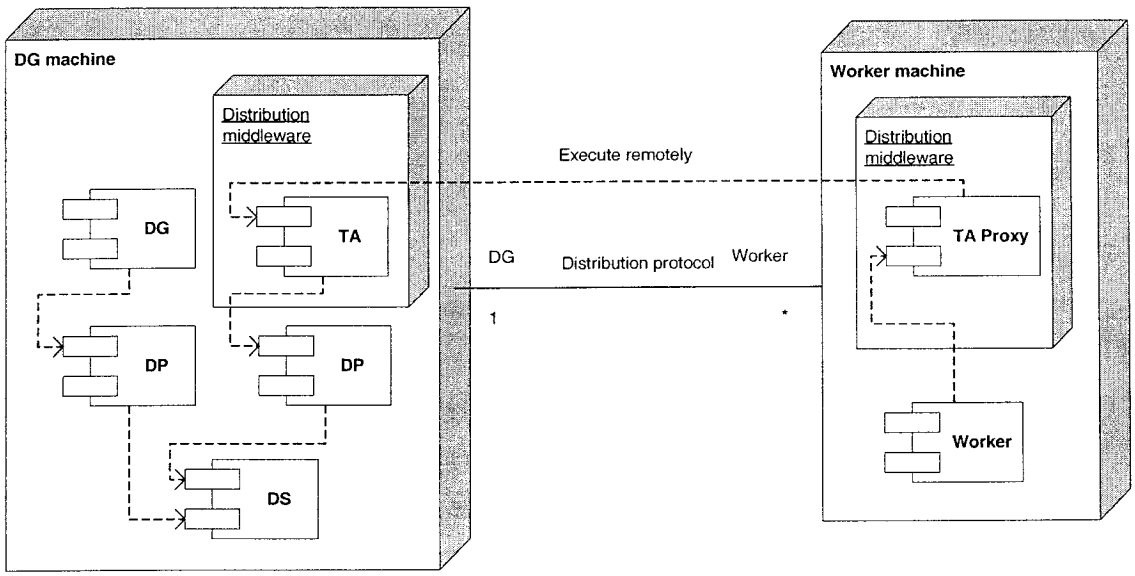


Fig.3.18. DMS Deployment Diagram "Local DD"

The *distribution unit* of the worker machine is the same like in the "Remote DD" case, i.e. it runs a worker, TA proxy and distribution middleware process. The distribution middleware process nests the TA proxy and makes the remote connection with the real TA possible.

There is another "Local DD" case where the DS resides the worker machine, i.e. the worker does not need a TA (respectively TA proxy) in order to access the DS. The structure of this *distribution unit* is similar to the structure of the DG machine in the "Local DD" case described above.

3.6. Summary

In this chapter, we have created a complete software design picture of our Demand Migration System (DMS). Our primary goal was to describe all the aspects of the

DMS, which necessitated the use of the "4+1" view model of software architecture [18]. In our DMS design model, we have used the following design views.

- A scenario view to illustrate how the DMS works, by presenting important scenarios of the system use.
- A logic view to illustrate the object model of our design. The object model consists of a high-level architectural view, where we have depicted the system's modules and their interfaces, and a low-level detailed design for some important modules that complete the DMS – Demand Dispatcher, JINI Transport Agent and JINI Library.
- A process view to illustrate the run-time view of our DMS in terms of processes and threads. Here, we have answered questions about non-functional requirements like reliability and accuracy, concurrency and consistency, scalability, upgradeability and heterogeneity.
- A deployment view to illustrate the structure of the run-time DMS configuration. With this view we have presented the physical distribution of our DMS among different machines.

In our DMS design we stuck to the basic design principles – loose coupling and high cohesion. We designed a system with loosely coupled components – the DMS components are independent standalone components that allow hot-plugging. There are no direct dependencies among the components. All the intercommunications are asynchronous and the components do not share common states. In addition, our design maximizes unit cohesion – the DMS' components are functional units where all the internal contributors work for performing the component functionality, which is exposed by the component's public interfaces.

Chapter 4: Implementation

> *The most common miracles of software engineering are the transitions from analysis to design and design to code.*
>
> Richard Due

This chapter provides a comprehensive implementation overview of our Demand Migration System. The implementation view corresponds to the detailed design provided in our design logical view (see section 3.3). Therefore, in the implementation view we provide an implementation-level description of the classes designed in the Demand Dispatcher package, JINI Transport Agent package and JINI Library package (see section 3.3). In addition, this chapter provides at the beginning an overview of the distributed technologies JINI, CORBA, DCOM and .Net Remoting, some of them being a base for our implementation solution. Finally, the chapter concludes with a presentation of other possible implementations for some of the DMS' components.

4.1. Distributed Technologies Aspects

Generally, the work done in the course of this thesis was influenced by the distributed technologies CORBA, DCOM, .NET Remoting and JINI [3, 4, 7, 13, 14, 16, 34, 42]. We investigated some aspects of these technologies with a great attention, especially those that took place in our design and implementation, like JINI and DCOM (as a next implementation solution). Before going over the DMS implementation, we analyze some of the most important aspects of these technologies.

4.1.1. JINI

JINI stands for Java Intelligence Network Infrastructure that is being developed by Sun Microsystems. JINI is an infrastructure for "federating services in a distributed system" [3]. JINI provides an open architecture for handling resource components - either hardware or software, within a network. The resource components are handled as services and the JINI systems provide mechanisms for their construction, lookup, communication, and use in a distributed system. JINI is a pure Java technology and integrates easily with the GIPSY, which is entirely implemented in Java. Therefore,

we chose JINI as a base for our first TA implementation – the JTA (see section 3.3.4 for the design and section 4.2.3 for the implementation). Despite the fact that JINI is independent of the network protocol, in our research and implementation we relied on a network, which is based on TCP/IP [34]. Therefore, we identify each JINI service uniquely in terms of a valid IP address and name.

4.1.1.1. Architecture

JINI manages and handles different available services by relying on a JINI Lookup Service (LUS) [3, 7, 34]. This service contains information about all other registered services in the JINI network. The clients may use the services by calling the LUS for discovering the registered services. JINI relies on a trio of internal protocols called *discovery*, *join*, and *lookup*, for performing the registration and use of the JINI-enabled components as JINI services [3, 7, 34]. A pair of these protocols - *discovery/join*, occurs when a service runs. *Discovery* occurs when a service looks for a LUS to register. *Join* occurs when a service has located a LUS and wishes to join it. *Lookup* occurs when a client or user needs to locate and invoke a service described by its interface type (written in Java). This process is the core of the JINI system. It can be referred to the *Discovery-Join-Lookup protocols* (see Fig.4.1).

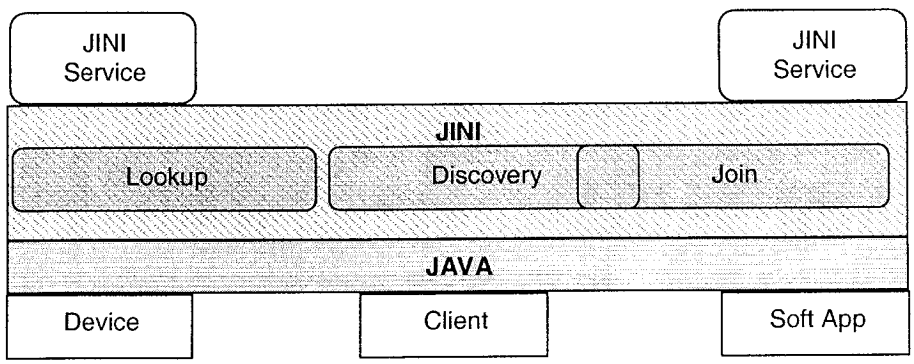


Fig.4.1. JINI Architecture

4.1.1.2. Features

Services in a JINI system communicate to each other by using a service protocol, which is a set of interfaces written in the Java programming language. The set of such protocols is open ended. The base JINI system defines a small number of such protocols, which define critical service interactions. Communication between services and clients can be accomplished by using Java Remote Method Invocation (RMI) [3,

7, 21]. RMI is a Java extension to the traditional Remote Procedure Call (RPC) mechanisms.

Discovery Process. The JINI discovery process is a process of finding lookup services. Both JINI services and clients use the *discovery* protocol, which exposes two discovery models - *multicast* discovery and *unicast* discovery [3, 7, 34]. Whereas, in *unicast* discovery the target is specified (for example by its IP address), in *multicast* discovery the service provider or service client broadcasts a message all around the network. The message identifies the service. This message should be delivered to all the lookup services within the net.

The JINI API exposes two utility classes that could be used for coding the discovery process – *LookupLocator* and *ServiceRegsitrar* (see Fig.4.2). Whereas, *LookupLocator* is used to get a reference to the JINI LUS, *ServiceRegistrar* holds that reference. *LookupLocator* searches for a LUS on any remote machine specified by an IP address. The method *getRegistrar()* returns the LUS reference if any.

```
LookupLocator lookup = new LookupLocator ("jini://192.168.8.11");
ServiceRegistrar registrar = lookup.getRegistrar ();
```

Fig.4.2. Coding Unicast Discovery Process

Join Process. The JINI join process is a process of registering a JINI service to a LUS. A special proxy service is sent from the service to the LUS (see Fig.4.3). This proxy implements the functionality that must be exposed to the clients, i.e. a JINI service implements part of its functionality in form of a proxy.

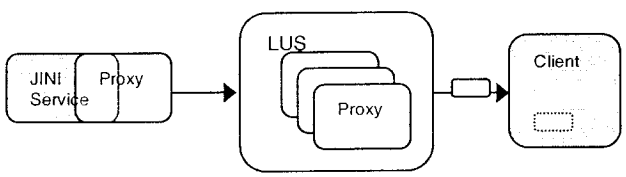


Fig.4.3. Discovery-Lookup-Join Protocols

For coding the join process we could use the *register()* method of the *ServiceRegistrar* class (see Fig.4.4). This method accepts as parameters an object instantiated from the JINI utility class *ServiceItem* and the lease time. The last

determines how long the registration will be available. The class *ServiceItem* accepts three parameters:

- id – universal unique identifier (UUID) for registered services (128-bit value). Service IDs are intended to be generated only by the lookup services, not by clients;
- service – the object implementing the actual JINI service;
- attrSets – service's attributes.

```
ServiceItem serviceItem = new ServiceItem (null, myServer, attr);
registrar.register (serviceItem, Lease.FOREVER);
```

Fig.4.4. Coding Join Process

Lookup Process. The JINI lookup process is a process of finding JINI services in the discovered LUS. This process is performed by the clients, which locate the needed JINI service by type. The type is a Java interface. If the service is found, the client application downloads the proxy and uses it locally (see Fig.4.3). The client invokes the service and interacts with it through the proxy service [34].

For coding the lookup process we could use the *lookup()* method of the *ServiceRegistrar* class (see Fig.4.5). This method accepts as a parameter an object instantiated from the JINI utility class *ServiceTemplate*. This template describes the JINI service, which is desired. The template acts like a query specification. The query could be by a service's id, by a service's class/interface name or by some attributes. The *lookup()* method returns either the proxy for the service, or null if no service matching the template is found.

```
ServiceTemplate template = new ServiceTemplate (null, null, aeAttributes);
myServerInterface = (IJINITransportAgent) registrar.lookup(template);
```

Fig.4.5. Coding Lookup Process

Distributed Leasing. Access to many of the services in the JINI system environment is lease based. A lease is a grant of guaranteed access over a time period. Each lease is negotiated between the user of the service and the provider of the service as part of the service protocol. Leases are either exclusive or non-

exclusive. Exclusive leases ensure that no one else may take a lease on the resource during the period of the lease. Non-exclusive leases allow multiple users to share a resource.

Distributed Transactions. A series of operations, either within a single service or spread among multiple services, can be wrapped in a transaction. The JINI transaction interfaces supply a service protocol needed to coordinate a *two-phase commit* [3, 7].

Distributed Events. The JINI architecture supports distributed events. An object may allow other objects to register interest in events in the object and receive a notification of the occurrence of such an event.

4.1.1.3. Object Persistence

To achieve object persistence JINI relies on the Java serialization architecture. The serialization architecture defines the interface *Serializable*, and instances of any Java class that implements this interface can be written into a byte stream or be read from a byte stream.

4.1.1.4. JavaSpace

In the course of this research, we found that one of the JINI applications – JavaSpace, is appropriate as a base for our Demand Space implementation. JavaSpace integrates the concept of tuple space [9]. The tuple space is the base for Ensemble Computing that joins the Distributed and Parallel Computing together [7]. The Ensemble Computing is based on the idea of existence of many active processes distributed over physically dispersed machines. Those machines "communicate by releasing data (a tuple) into a common tuple space" [7].

JavaSpace extends that model by defining the data as real objects, i.e. we pass not only data but also object's behavior (see Fig.4.6). The use of RMI and object serialization makes passing of live objects possible. In addition, JavaSpaces implements features like distributed events, leasing and lightweight transactions.

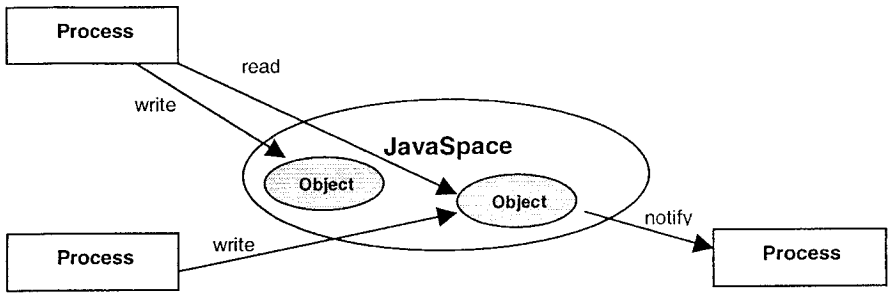


Fig.4.6. JavaSpace Model

JavaSpace Interface. Any JavaSpace instance is accessible via an API Java interface called *JavaSpace*. This interface exposes the methods *read()*, *write()*, *take()*, *notify()*, *readIfExists()* and *takeIfExists()*. These methods allow clients to perform the basic operations on a JavaSpace instance [7]. The *getSpace()* method helps in gaining access to an instance of JavaSpace (see Fig.4.7).

```
JavaSpace space = getSpace();
```

Fig.4.7. Coding JavaSpace Access

4.1.2. CORBA

CORBA stands for Common Object Request Broker Architecture that is being developed by the Object Management Group (OMG) [42]. CORBA is a standard architecture for distributed object systems. CORBA specifies a system that provides interoperability between objects in a heterogeneous, distributed environment and in a way transparent to the programmer.

4.1.2.1. Architecture

CORBA defines architecture for distributed objects. The CORBA architecture is based on services and the access to the CORBA services is via static and dynamic interfaces. Those interfaces are implemented in the form of stubs and skeletons. CORBA relies on the stubs to send requests from clients, and relies on the skeletons

to receive and forward requests to objects. The following figure illustrates the primary components in the CORBA architecture.

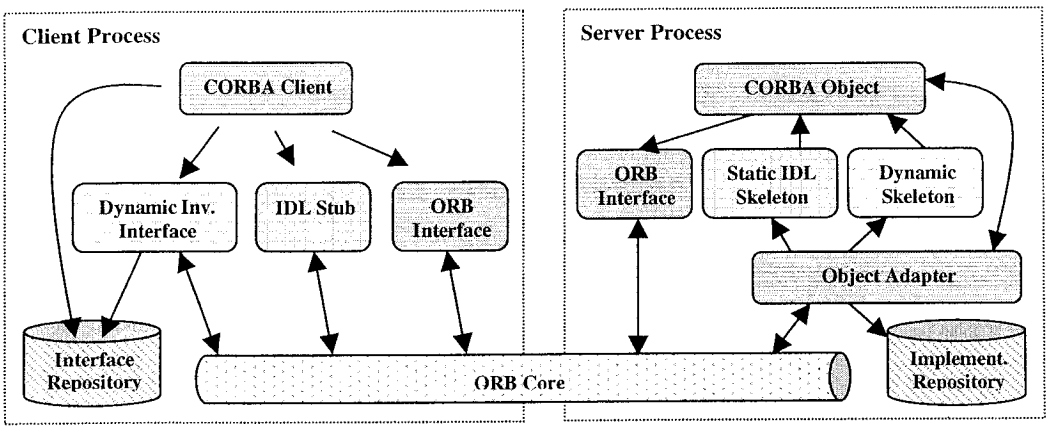


Fig.4.8. CORBA Architecture

CORBA Client and Object. The CORBA client is the program entity that invokes operations on a CORBA object that is a CORBA programming entity. CORBA makes the access to the services of a remote object transparent to the client.

Object Request Broker (ORB). The ORB is a CORBA distributed service that performs the request-result operation between a CORBA object and CORBA client [42]. The ORB locates the remote object on the network, communicates the request to the object, waits for the results and when available communicates those results back to the client. It acts like a transport channel between the client and remote object. The ORB implements location transparency, the client and the CORBA object use exactly the same request mechanism, regardless of where the object is located.

Dynamic Invocation Interface and Dynamic Skeleton. The Dynamic Invocation Interface (DII) allows the client to specify requests to objects, which definition and interface are unknown at the client's compile time (late banding) [13, 42]. In order to use the DII, the client has to compose a request, including the object reference, the operation and a list of parameters. These specifications - of objects and services they provide, are retrieved from the Interface Repository (see Fig.4.8). The DII works with the Dynamic Skeleton on the server side. The Dynamic Skeleton allows an ORB to deliver requests to an object implementation that does not have compile-time knowledge of the type of the object it is implementing.

IDL Stub and IDL Skeleton. The IDL stub helps for the mapping of a non-object-oriented language. The stubs make calls on the rest of the ORB using interfaces that are private to, and presumably optimized for, the particular ORB Core. If more than one ORB is available, there may be different stubs corresponding to the different ORBs. The IDL stubs usually work with IDL skeletons (see Fig.4.8). The Static IDL skeleton is generated by CORBA skeleton, which is used in CORBA static invocation.

Object Adapter. The Object Adapter (OA) performs the communication between the object implementation and the ORB core. It handles services such as generation and interpretation of object references, method invocation, security of interactions, object and implementation activation and deactivation, mapping references corresponding to object implementations and registration of implementations [13, 42].

ORB Interface. The ORB Interface is an interface that goes directly to the ORB (see Fig.4.8). It is the same for all the ORBs and does not depend on the object's interface or object adapter. Because most of the functionality of the ORB is provided through the object adapter, stubs, skeleton, or dynamic invocation, there are only a few operations that are common across all objects.

Repositories. The Interface Repository is a database that provides a persistent storage mechanism of object interface definitions. The Implementation Repository is a service that allows the ORB to locate and activate implementations of objects.

4.1.2.2. Features

The services that an object provides are given by its interfaces. Interfaces are defined in an Interface Definition Language (IDL) [13, 42]. Distributed objects are identified by object references, which are presented by IDL interfaces. Fig.4.9 depicts an object request, and the object reference is presented by an IDL interface.

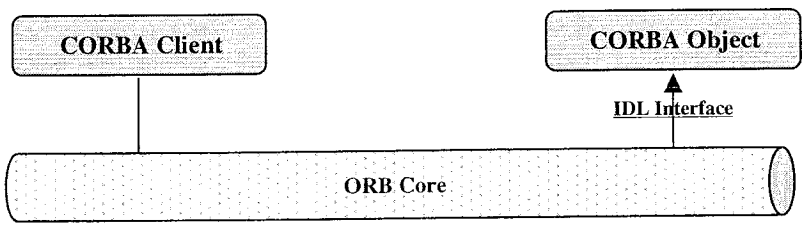


Fig.4.9. Client-Object Request

CORBA Services. CORBA exposes a set of distributed services to support the integration and interoperation of distributed objects. As depicted by Fig.4.10, the services, known as CORBA services (COS), are defined on top of the ORB. They are standard CORBA objects with IDL interfaces.

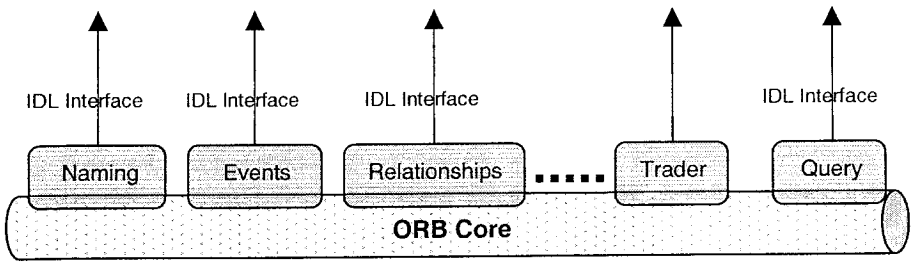


Fig.4.10. CORBA Services

The following elements describe some of the services exposed by CORBA:

- Object life cycle - defines how CORBA objects are created, removed, moved, and copied.
- Naming - defines how CORBA objects can have friendly symbolic names.
- Events - decouples the communication between distributed objects.
- Relationships - provides arbitrary typed n-ary relationships between CORBA objects.
- Externalization - coordinates the transformation of CORBA objects to and from external media.
- Transactions - coordinates atomic access to CORBA objects.
- Concurrency control - provides a locking service for CORBA objects in order to ensure serializable access.
- Property - supports the association of name-value pairs with CORBA objects.
- Trader - supports the finding of CORBA objects based on properties describing the service offered by the object.
- Query - supports queries on CORBA objects.

How do remote invocations work? In order to invoke a remote object instance, a client first obtains its object reference by relying either on the *naming* service or *trader* service. The client invokes the remote and local objects in the same way. When the ORB examines the object reference and discovers that the target object is

remote, it routes the invocation out over the network to the remote object's ORB (see Fig.4.11).

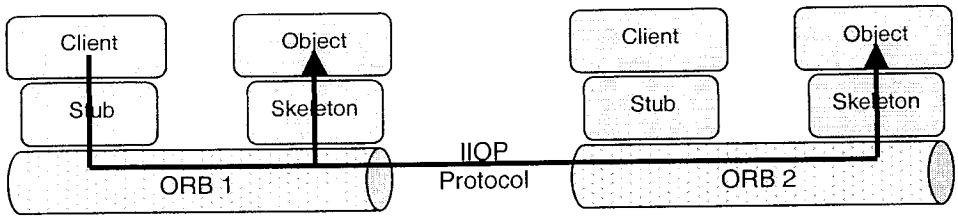


Fig.4.11. Remote and Local Invocation

For the remote invocation, the client's ORB and object's ORB must agree on a common communication protocol. The ORB uses as a standard the Internet inter-ORB Protocol (IIOP) [13, 42].

4.1.2.3. Object Persistence

CORBA provides a service called Persistent State Service (PSS) that enables object persistence for CORBA objects. The PSS exposes two store mechanisms. The objects are stored as *storage types* in so-called *storage homes*, and *storage homes* are themselves stored in *datastores* [25, 4]. The *datastores* are entities for managing data in persistent store mechanisms like databases and files. There are two ways to define the datastore schema and the interface of the *storage type*:

- In the first way, the PSS relies on the Persistent State Definition Language (PSDL), which is the standard IDL language for persistence objects. The PSDL defines the *storage homes* and the *storage types* as *storagehome* and *storagetype* constructs.
- The second way is to use the transparency mechanisms provided by the CORBA compliant development language. This is known as Transparent Persistence [25].

In addition to the PSS, CORBA exposes a service called CORBA Externalization Service. That service standardizes interfaces that enable CORBA objects to write their states into a sequential byte stream and to read their states from such a stream.

4.1.3. DCOM

DCOM is the distributed extension of COM, which states for Component Object Model. COM is being developed by Microsoft as a general software architecture that provides a framework for integrating software components. This framework allows developers to build systems by assembling reusable components. By defining an application-programming interface (API), COM allows the creation and integration of components in custom applications or allows diverse components to interact [14, 27].

We are planning implementation of our second TA based on COM/DCOM. Hence, in the course of this research we investigated deeply the COM aspects.

4.1.3.1. Architecture

COM defines how components and their clients interact. This interaction is defined such that the client and the component can connect without the need of any intermediary system component. The client calls methods in the component without any additional overhead. The COM architecture exposes two principal aspects:

- COM uses globall unique identifiers called *class identifiers* (CLSIDs) - to uniquely identify each COM object. CLSIDs are 128-bit integers that are guaranteed to be "unique in the world across space and time" [26].
- COM objects interact with each other and with the system through a collection of interfaces. A client application has access to the object's services only through the object's set of interfaces (see Fig.4.12). These interfaces adhere to a binary structure, which provides the basis for interoperability between software components written in arbitrary languages.

Fig.4.12. COM Interaction

A COM interface is a strongly-typed contract between software components that provides a small but useful set of semantically related operations. A COM object can support any number of interfaces. Fig.4.12 depicts the interfaces as small circles connected to the COM object.

The COM architecture requires every COM object to run inside of a process server, called also COM component (see Fig.4.13). A single server can support multiple COM objects. COM objects are either implemented within executables (EXEs) or within Dynamic Linked Libraries (DLLs). COM objects implemented in EXEs are called *out-of-process* servers, and these implemented in DLLs are called *in-process* servers [14, 26].

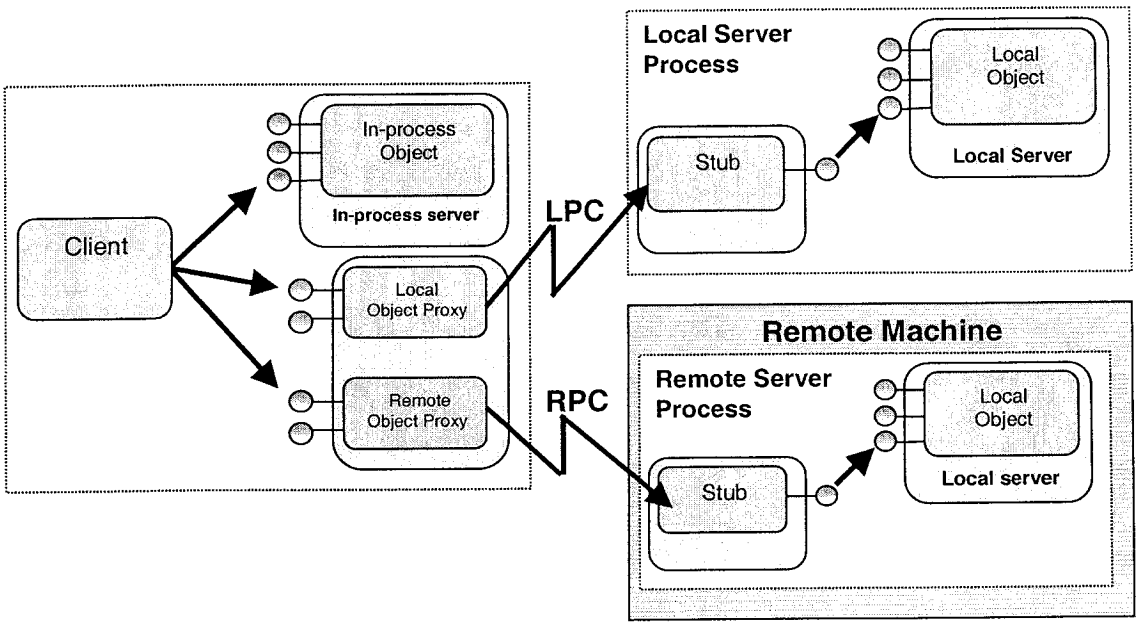


Fig.4.13. DCOM Architecture

The *out-of-process* servers are *local* when run on the client's machine, or *remote* when run on a remote machine. DCOM is about *remote* COM components. Whereas, the *in-process* COM objects run in the client's address space, the *out-of-process* COM objects are shielded in a separate process. A client that needs to communicate with a component in another process cannot call the component directly, but has to use some form of inter-process communication provided by the operating system. Such a communication is called Local Procedure Call (LPC) for the *local* process server and Remote Procedure Call (RPC) for the remote one (see Fig.4.13). COM provides this communication in a completely transparent fashion: it intercepts calls from the client and forwards them to the component in another process. For this transparency COM relies on client-side proxies and server-side stubs that internally perform the complex inter-process communication. When the client and component reside on different machines, DCOM simply replaces the local inter-process communication with a network protocol.

4.1.3.2. Features

Regardless of the kind of the server in use (in-process, local, or remote), the COM client always asks COM to instantiate objects in exactly the same manner. There are three COM object invocation steps.

Client Request. The client request is the first COM object invocation step. The COM client is an application that invokes the COM API in order to instantiate a new COM object. It passes a CLSID to COM, and asks for an instantiated object in return [14]. The client request includes two specific tasks:

- The COM client must verify that the COM Library version is new enough to support the functionality expected by the application. In general, an application can use an updated version of the library, but not an older one.
- The COM client must initialize the COM Library.

There are two ways for making a request. The simplest one is to call the COM function *CoCreateInstance* that creates an object by a given CLSID, and returns an interface pointer of any requested type. Alternately, by calling *CoGetClassObject,* the client can obtain an interface pointer to what is called the *class factory* object for a CLSID (Fig.4.14). This class factory supports an interface called *IClassFactory* through which the client asks the factory to manufacture an object of its class [27].

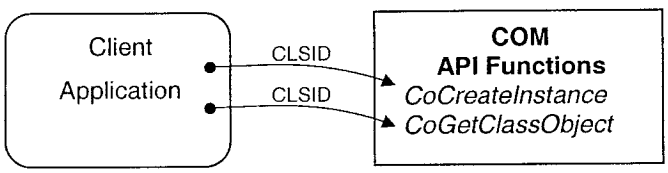


Fig.4.14. COM Client Request

Server Location. The server location is the second COM object invocation step. In this step, COM locates the object implementation and initiates a server process for the object. A special component called Service Control Manager (SCM) is responsible for the location and execution of the COM server that implements the COM object. The SCM ensures that when a client request is made, the appropriate server is connected and ready to receive the request. The actions, undertaken by the SCM, depend on the kind of the COM server:

- *In-Process* - The SCM returns the file path of the DLL containing the object server implementation. The COM library then loads the DLL, and asks it for its class factory interface pointer.
- *Local* - The SCM finds and starts the local EXE, which registers a class factory interface pointer.
- *Remote* - The local SCM contacts the SCM running on the appropriate remote computer, and forwards the request to the remote SCM. The remote SCM obtains a class factory interface pointer in one of the two ways described above (in-process or local). The remote SCM then maintains a connection to that class factory, and returns a RPC connection to the local SCM [27].

Object Creation. The third invocation step, called object creation, creates the object by a given CLSID. It involves three internal steps (see Fig.4.15):

- Obtains the *class factory* for the CLSID.
- Asks the *class factory* to instantiate an object of the class, and returns an interface pointer to the COM client.
- Initializes the COM object.

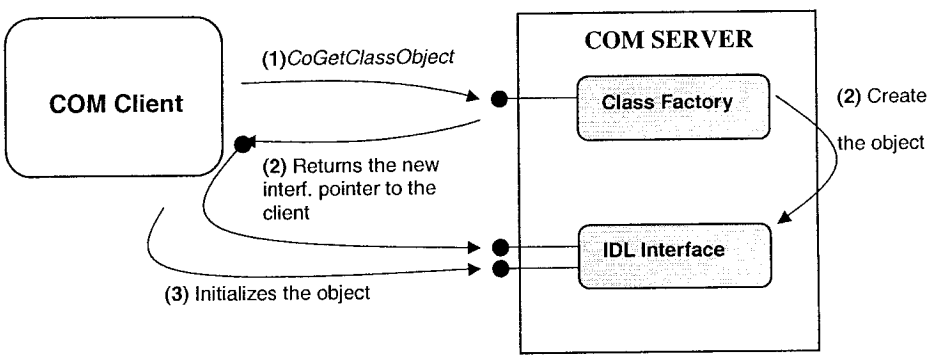


Fig.4.15. COM Object Creation

4.1.3.3. Object Persistence

COM performs object persistence via an interface called *IPersist* [4, 27]. In addition, COM extends the *IPersist* interface into the *IPersistStream* interface. That interface supports serialization of objects into a byte stream that can be stored in a file. The last should be instantiated from a class implementing the interface *IPersistentInterface.*

4.1.4. .Net Remoting

Microsoft .NET Remoting was introduced by Microsoft with the advent of .NET and .NET Framework [16, 28]. .NET is a new distributed technology that allows programs and software components to interact across application domains, processes, and machine boundaries. Remoting allows you to pass objects or values across servers in different domains using several different protocols.

4.1.4.1. Architecture

.NET Remoting uses a flexible and extensible architecture. Remoting uses the .NET concept of the Application Domain (AppDomain) to determine its activity [16, 28]. An AppDomain is an abstract construct for ensuring isolation of data and code, but not having to rely on operating system specific concepts such as processes or threads. .NET Remoting relies on a set of internal contributors for performing the remote communication between the client and server (see Fig.4.16). The client and the server communicate in a transparent manner. The client calls a remote method, which call is received by a *proxy*. The *proxy* looks just like the real server as it has the same public methods. The *proxy* just converts the method call to a message so that it can be sent across the network. Next, the message is passed to a *formatter.* The *formatter* is used for encoding and decoding the message before and after it is transported by a *transport channel* (see Fig.4.16). Finally, the message is delivered to the server via a *dispatcher.* The server invokes the method and once the execution is completed, the process is reversed and the results are returned back to the client.

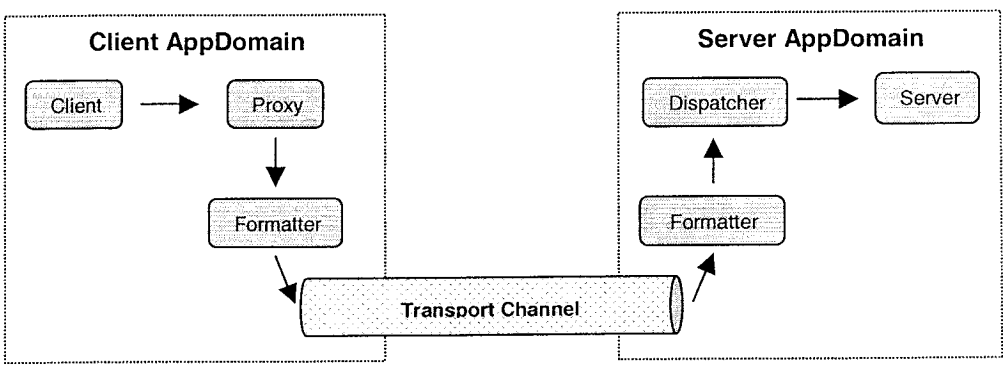


Fig.4.16. .Net Remoting Architecture

The formatters are able to perform *binary* and *XML* encoding. Whereas, the binary encoding is appropriate when performance is critical, the XML [29] encoding is appropriate when interoperability with other remoting frameworks is essential.

A transport channel provides the underlying transport mechanism for the communication between the client and server.

4.1.4.2. Features

.Net Remoting migrates the objects between so called *remotable* objects. *Remotable* are objects distributed across different platforms. In general, there are two types of *remotable* objects - Marshall-By-Value and Marshall-By-Reference objects.

Marshall-By-Value. Marshall-By-Value (MBV) objects are copied and passed over the server application domain to the client application domain. The MBV objects are initially located on the server. However, when the client invokes a method of the MBV object, the MBV object is serialized (by the Remoting Framework), migrated over the network (using the transport channel) and restored on the client as an exact copy of the server-side object. The Method is then invoked directly on the Client.

Marshall-By-Reference. Marshall-By-Reference (MBR) objects are accessed on the client side by using a proxy. MBR objects are remote objects they always reside on the server and all the methods invoked on these objects are executed at the server side. The client communicates with the MBR objects on the server using the local proxy object that holds a reference to the MBR object.

Leased Based Lifetime. For the MBR objects, having object references that are transported outside the application, a lease is created. The lease controls the lifetime of the object. The lease time could be set to infinity.

4.1.4.3. Object Persistence

.Net Remoting makes the objects persistent by declaring their class *serializable*, or by implementing the *ISerializable* interface. All objects that have to cross the application domain boundary have to implement the *ISerializable* interface. When the object is passed as a parameter, the framework serializes the object and transports it to the destination AppDomain, where the object will be reconstructed.

4.2. Current Implementation

In this section, we provide some sample code with a brief description for some of the important DMS' components. For the implementation of our DMS, we adopted the Java programming language, this being the implementation language for the other GIPSY' parts. Our current implementation is based on JINI, and we relied on the Java and JINI API for coding our DMS.

4.2.1. JINI Library

The following elements describe the implementation of the JINI Library package (see section 3.3.2).

Class *JINILibrary*. The class *JINILibrary* is the main class for the JINI Library package (see section 3.3.2.1). This class implements the *IJINILibrary* interface. Listing 4.1 depicts the implementation of the *findRegistrar()* and *findDMSService()* methods.

```
public ServiceRegistrar findRegistrar (String sIPAddress)
        throws LUSException {
        LookupLocator lookup = null;

//**** Prepare for Unicast Discovery - get the LUS
        lookup = new LookupLocator ("jini://" + sIPAddress);
        if (lookup == null)   throw new LUSException(sIPAddress);
         theRegistrar = lookup.getRegistrar();
        return theRegistrar;
}

 public ServiceRegistrar findRegistrar (String discGroupNames) {
//**** Search for the GIPSY LUS
        disco = new LookupDiscovery(discGroupNames);

//**** Run discovery multicast protocol via the listener
        disco.addDiscoveryListener(new ServiceListener ());
        while (theRegistrar == null) ;
        return theRegistrar;
}
 public Object findDMSService (ServiceTemplate template)  {
        Object oDMSService = null;
        if (theRegistrar==null) theRegistrar = findRegistrar();
        try  {
                oDMSService = theRegistrar.lookup(template);
        }
        catch (RemoteException e)  {
                throw new JINILibraryException ("Remote error: " + e.getMessage());
        }
        return oDMSService;
}
```

Listing 4.1. Coding *JINILibrary* Class Methods

Whereas, the first *findRegistrar()* method performs *unicast discovery*, the second performs *multicast discovery* (see section 4.1.1.2). The second method relies on the *ServiceListener* class for performing asynchronous event-driven LUS discovery. The method *findDMSService()* searches in the LUS, held by the class data filed *theRegistrar*, for a registered DMS service that matches with the template. This method is used by the DGs and workers in order to connect with a JTA (see section 3.3.4.1).

Class *ServiceListener*. This class is designed to perform *discovery* protocol (see section 3.3.2.2 and section 4.1.1.1). The class *ServiceListener* is implemented as an inner class for the class *JINILibrary*. The class simply implements the JINI *DiscoveryListener* interface, which has two methods – *discovered()* and *discarded()*.

The following implementation was inspired by W. Edwards [3].

```
public void discovered(DiscoveryEvent ev) {
    ServiceRegistrar[] newregs = ev.getRegistrars();
    for (int i=0 ; i<newregs.length ; i++) {
        if (!registrations.containsKey(newregs[i]))
            assignRegistrar(newregs[i]);
    }
}

public void discarded(DiscoveryEvent ev) {
    ServiceRegistrar[] deadregs = ev.getRegistrars();
    for (int i=0 ; i<deadregs.length ; i++) {
        registrations.remove(deadregs[i]);
    }
}
```

Listing 4.2. Coding *ServiceListener* Class Methods

The *discovered()* method calls the method *assignRegistrar()* of the *JINILibrary* class when a new LUS is found. In addition, it maintains a hash table that registers all the LUS we have found. Whenever the *discovery* protocol informs for new LUSs, we fetch them out of the *DiscoveryEvent* and call the *assingRegistrar()* method. This implementation assures that the *JINILibrary* class will be asynchronously notified whenever a LUS is found, i.e. the notification is done outside the main flow of control, which is not blocked waiting the end of the *discovery* process.

4.2.2. Demand Dispatcher

In the course of this implementation, one of the principle questions was about the implementation of the Demand Space (DS). Since we decided to integrate an already existing Object Oriented Database (OOD), the questions were about the right choice and integration. Our thorough investigation into JINI helped in our choice of JavaSpace as a base for our Demand Space implementation (see section 4.1.1.4). JavaSpace is not an OOD, although it does have OOD characteristics. JavaSpace does not define a query language. Instead, objects are retrieved with templates. In addition, JavaSpace provides a simple API and defines a lightweight persistence engine [7]. The advantage lies in the simple integration and programming model. Hence, the choice of JavaSpace simplified the implementation of the Demand Dispatcher (DD). The following elements describe the implementation of the Demand Dispatcher package (see section 3.3.3).

Class *DemandState*. The class *DemandState* represents an enumeration type used to distinguish the demands by their state. It defines the three states – *pending*, *in process* and *computed* and implements functions for determining the state of a demand. Listing 4.3 depicts the full implementation of the class.

```
public class DemandState implements Serializable {
        private static final String STR_PENDING = "pending";
        private static final String STR_INPROCESS = "inprocess";
        private static final String STR_COMPUTED = "computed";
        private String sState = "";
        public static final DemandState PENDING = new DemandState(STR_PENDING);
        public static final DemandState INPROCESS = new DemandState(STR_INPROCESS);
        public static final DemandState COMPUTED = new DemandState(STR_COMPUTED);

        public boolean isPending() {
                if (sState.compareTo(STR_PENDING) == 0) return true;
                else return false;
        }
        public boolean isInProcess() {
                if (sState.compareTo(STR_INPROCESS) == 0) return true;
                else return false;
        }
        public boolean isComputed() {
                if (sState.compareTo(STR_COMPUTED) == 0) return true;
                else return false;
        }
        private DemandState (String newState) {
                sState = newState.toLowerCase();
        }
}
```

Listing 4.3. Coding *DemandState* Class

Class *DispatcherEntry*. The class *DispatcherEntry* unifies the demands and results (*computed* demands) as one unit (see section 3.3.3.2). The class instantiates entries those are able to be stored in the Demand Space. Since our DS implementation is based on JavaSpace, the class *DispatcherEntry* extends the *AbstractEntry* JavaSpace's class. This class extends the *Serializable* interface and makes the entries storable in JavaSpace [7].

Listing 4.4 depicts the full implementation of the class.

```
public class DispatcherEntry extends AbstractEntry {
        public Uuid oUniqueID;
        public DemandState dState;
        public Serializable oObject;

        public DispatcherEntry () {
          this(null);
        }

        public DispatcherEntry (Uuid theID)          {
          this(theID, null);
        }

        public DispatcherEntry (Uuid theID, Serializable theObject) {
//**** The demand's state is pending par default
          this(theID, theObject, DemandState.PENDING);
         }

        public DispatcherEntry (Uuid theID, Serializable theObject, DemandState newState) {
          oUniqueID = theID;
          oObject = theObject;
          dState = newState;
        }
}
```

Listing 4.4. Coding *DispatcherEntry* Class

The class implements only constructors, including the mandatory default no-argument constructor (see section 3.3.3.2). In addition, the class defines three data fields holding the demand, demand's GUID and demand's state.

Class *DispatcherProxy*. The class *DispatcherProxy* is the main class for the Demand Dispatcher package (see section 3.3.3.2). This class implements the *IDemandDispatcher* interface, which is the public interface of the package (see Fig.3.12). The methods exposed by this interface form an abstraction layer on top of JavaSpace. Therefore, the class *DispatcherProxy* implements the functions necessary for operating on JavaSpace.

Listing 4.5 depicts the implementation of the *write()* method, which is used by the *IDemandDispatcher*'s methods *writeDemand()* and *writeResutl()*. This method wraps a demand, passed as a parameter, in a *DispatcherEntry* object. In addition, it checks the demand's status, and if is *computed* generates a GUID for the demand.

```
private synchronized static Uuid write(Uuid theID, Serializable theObject, DemandState theState)
  throws DemandDispatcherException {
        try {
          Lease theLease;
          Uuid oUniqueID;
          DispatcherEntry theEntry;

          if (theState.isComputed())  oUniqueID = theID;
          else  oUniqueID = getNewUniqueID();

          theEntry = new DispatcherEntry (oUniqueID, theObject, theState);
          theLease = oJavaSpace.write(theEntry, null, Lease.FOREVER);

          return theEntry.oUniqueID;
        }
        catch(Exception ex) {
          throw new DemandDispatcherException (ex.getMessage());
        }
}
```

Listing 4.5. Coding *DispatcherProxy* Class – *write*() Method

Listing 4.6 depicts the implementation of the *read()* method, which is used by the *IDemandDispatcher*'s methods *readDemand()* and *readResutl()*. The method searches JavaSpace for an entry that matches the template provided as a *DispatcherEntry* variable *theEntry*. The template matches any entry that has an ID equals to *theID* and a state equals to *theState*. The *read()* method uses the JavaSpace's *read()* method to perform the search request. This request will block the current thread until a matching entry is found.

```
private synchronized static DispatcherEntry read(Uuid theID, DemandState theState)
  throws DemandDispatcherException {
        try {
          DispatcherEntry theEntry;

          theEntry = new DispatcherEntry (theID, null, theState);
          theEntry = (DispatcherEntry) oJavaSpace.read(theEntry, null, Lease.FOREVER);

          return theEntry;
        }
        catch(Exception ex) {
          throw new DemandDispatcherException (ex.getMessage());
        }
}
```

Listing 4.6. Coding *DispatcherProxy* Class – *read*() Method

JavaSpace is a JINI service. Therefore, the process to access JavaSpace is like accessing any other JINI service. Listing 4.7 depicts the implementation of the *getJavaSpace()* method, which is used by the class to access JavaSpace. The method refers to the *JINILibrary*'s *findRegistrar()* method to perform the *unicast discovery* protocol.

```
private synchronized static JavaSpace getJavaSpace(String hostName, String spaceName) {
        try {
                LookupLocator lookup = null;
                ServiceRegistrar registrar = null;
                Entry entries[] = {new Name(spaceName)};
                JavaSpace theSpace = null;

                Class[] types = new Class[] { JavaSpace.class };
                ServiceTemplate template = new ServiceTemplate(null, types, entries);

//**** Set the security manager
                if (System.getSecurityManager() == null)
                        System.setSecurityManager(new RMISecurityManager());

                registrar = JINILibrary.findRegistrar(hostName);
                theSpace = (JavaSpace)registrar.lookup(template);

                return theSpace;
        }
        catch (Exception ex) {
                printOutError(ex.getMessage());
                return null;
        }
}
```

Listing 4.7. Coding *DispatcherProxy* Class – *getJavaSpace*() Method

Listing 4.8 depicts the implementation of the *cancelDemand()* method, which cancels any demand holding a specified ID. The method uses the JavaSpace's *take()* method, which removes the matching entry from the space.

```
public synchronized void cancelDemand(Uuid theID)
  throws DemandDispatcherException {
        try {
          DispatcherEntry theEntry;

          theEntry = new DispatcherEntry (theID, null, null);
          theEntry = (DispatcherEntry) oJavaSpace.take(theEntry, null, Lease.FOREVER);

        }
        catch(Exception ex) {
          throw new DemandDispatcherException (ex.getMessage());
        }
}
```

Listing 4.8. Coding *DispatcherProxy* Class – *cancelDemand*() Method

Threading. In our implementation, the *DispatcherProxy* instances run in a separate thread of control (see Fig.3.16), due to the fact that most of the JavaSpace operations involve remote calls, which may take a relatively long amount of time to return. For example, the *read()* and *write()* methods could block the execution for an infinite amount of time, since the DP- DS communication is synchronous (see section 3.4.2).

4.2.3. JINI Transport Agent

In the course of this implementation we were guided not only by our JINI Transport Agent (JTA) design model (see section 3.3.4), but also by two technologies that influenced tremendously our work. These technologies are JINI and RMI. Whereas, JINI is the implementation base for our solution, RMI is convenient way to implement remote communication between two Java objects (respectively two JINI services) (see section 4.1.1.2). The JTA implementation as a JINI service was inspired by W. Edwards [3].

Class *JINITransportAgent*. The class *JINITransportAgent* is the main class for the JTA package (see section 3.3.4). The class implements the *Runnable* Java interface and causes a simple thread to be started. This thread simply sleeps, but keeps the JTA alive as long as the thread is. The *main()* method creates a *JINITransportAgent* instance and starts the background thread to keep the application alive. In addition, the class implements functions allowing connecting the JTA with the DD, creating the JTA proxy and registering the JTA on the LUS (see section 3.3.4.1). The class *JINITransportAgent* is implemented as a wrapper class that defines one nested interface and two classes. The first of these is the *IJTABackendProtocol* interface, which is used by the JTA proxy to communicate with the remote back-end object (see section 3.3.4.2). This interface extends the *Remote* RMI interface, i.e. it allows communication between objects in different Java Virtual Machines (JVMs). Since the interface is only used by the proxy and by the back-end remote object, it is an implementation detail of the JTA and it should be hidden from clients. The other two classes nested by the *JINITransportAgent* class are the *JTABackend* class and *JINITransportAgentProxy* class.

Listing 4.9 depicts the implementation of the class' constructor. The constructor sequentially creates the JTA proxy, sets the security manager, connects the JTA with the DD and registers the JTA on the found LUS. The security manager ensures that any class that is loaded remotely does not perform illegal operations. In addition, the constructor initializes some data fields. The data field *sCodebase* holds the code location that will be provided to the clients in order to locate and download the proxy. The data field *lookupItem* holds the item to be registered on the discovered LUS (see the *registerWithLookup()* method below).

```
protected final String [] DISCOVERY_GROUP_NAMES = {"gipsy"};
protected static String sCodebase = "";
protected ServiceItem = lookupItem;
.....
public JINITransportAgent ()
    throws IOException, ClassNotFoundException, JTAException {
     try  {
              ServiceRegistrar oRegistrar;
              sIPLocalAddress = GetLocalIPAddress();
              sCodebase = "http://" + sIPLocalAddress + ":8085/";
              lookupItem = new ServiceItem (null, createProxy(), null);

//**** Set a security manager
              if (System.getSecurityManager() == null)
                       System.setSecurityManager(new RMISecurityManager());

//**** Connect with the Demand Dispatcher
              connectDemandDispatcher();

//**** Run multicast discovery protocol and register with the found LUS
              oRegistrar = JINILibrary.findRegistrar(DISCOVERY_GROUP_NAMES);
              registerWithLookup (oRegistrar);
     }
     catch (Exception e)  {
              throw new JTAException (e.getMessage());
     }
    }
```

Listing 4.9. Coding *JINITransportAgent* Class – Constructor

Listing 4.10 depicts the implementation of the *registerWithLookup()* method, which registers the JTA on the discovered LUS.

```
protected synchronized void registerWithLookup(ServiceRegistrar registrar)
        throws JTAException {
         ServiceRegistration registration = null;
         try { registration = registrar.register(lookupItem, LEASE_TIME);}
         catch (RemoteException e) {
                  throw new JTAException ( e.getMessage()); }
         if (item.serviceID == null)
                  lookupItem.serviceID = registration.getServiceID();
         registrations.put(registrar, registration);
         startAfterRegistration();
    }
```

Listing 4.10. Coding *JINITransportAgent* Class – *registerWithLookup*() Method

Listing 4.11 depicts the implementation of the *createProxy()* method, which creates the JTA proxy. This method does the following:

- First, the method creates a back-end object and makes sure it is known to the RMI activation system.
- Second, the method creates a proxy with a reference to the back-end object. This allows the proxy to call back to it.
- Third, the method returns the proxy to the caller.

In addition, the method sets a security policy that the activation group running the back-end service will use. The security policy is provided in a policy file.

```
protected static final String SECURITY_POLICY_FILE = "activation.policy";
...
protected IJINITransportAgent createProxy() {
    try  {
        Properties props = new Properties();
        props.put("java.security.policy", SECURITY_POLICY_FILE);
        ActivationGroupDesc group = new ActivationGroupDesc(props, null);

        ActivationGroupID gid = ActivationGroup.getSystem().registerGroup(group);
        printOut("2.Has been registering the group and get the ID");

        ActivationGroup.createGroup(gid, group, 0);

        String location = sCodebase;
        MarshalledObject data = null;
        ActivationDesc desc = new ActivationDesc("gipsy." +
                "JINITransportAgent$JTABackend", location, data);

        JTABackendProtocol backend = (JTABackendProtocol) Activatable.register(desc);

        return new JINITransportAgentProxy(backend);
    }
    catch (RemoteException e) {
        printOutError("Error creating backend object: " + e.getMessage());
        return null;
    }
    catch (ActivationException e) {
        printOutError("Problem with activation: " + e.getMessage());
        e.printStackTrace();
        return null;
    }
}
```

Listing 4.11. Coding *JINITransportAgent* Class – *createProxy*() Method

Class *JTABackend.* The *JTABackend* class instantiates back-end objects. This class implements the *IJTABackendProtocol* interface and extends the RMI's class *Activatable.* The last makes the class' methods callable from remote JVMs. The implementation of the *IJTABackendProtocol* interface determines how the proxy will

communicate with the back-end. The following listing depicts the implementation of this interface.

```
public JTABackend (ActivationID id, MarshalledObject data)
        throws RemoteException  {
        super(id, 0);
}
public synchronized IWorkDemand fetchDemand(Uuid idDemand, String sSenderIP)
        throws RemoteException, DemandDispatcherException {
        DispatcherEntry oEntry = null;
        IWorkDemand oDemand;
        oEntry = oDemandDispatcher.readDemand();
        oDemand = (IWorkDemand)oEntry.oObject;
        return oDemand;
}

public synchronized IWorkResult fetchResult(Uuid idResult, String sSenderIP)
        throws RemoteException, DemandDispatcherException {
        DispatcherEntry oEntry = null;
        IWorkResult oResult;
        oEntry = (DispatcherEntry)(oDemandDispatcher.readResult(idResult));
        oResult = (IWorkResult)oEntry.oObject;
        return oResult;
}

public synchronized Uuid carryDemand(IWorkDemand oDemand, String sSenderIP)
        throws RemoteException, DemandDispatcherException {
        Uuid oUniqueID;
        oUniqueID = oDemandDispatcher.writeDemand(oDemand);
        return oUniqueID;
}

public synchronized Uuid carryResult(IWorkResult oResult, Uuid idResult, String sSenderIP)
        throws RemoteException, DemandDispatcherException {
        oDemandDispatcher.writeResult(idResult, oResult);
        return idResult;
}
```

Listing 4.12. Coding *JTABackend* Class – *IJTABackendProtocol* Implementation

The TA proxy calls remotely the methods listed above. These methods use internally the *oDemandDispatcher* data filed. This data field belongs to the class *JINITransportAgent*, and holds a reference to an instance of the *DispatcherProxy* class (see section 4.2.2). Therefore, the TA proxy calls the methods of the *DispatcherProxy* via the *JTABackend* instance. The *JTABackend* class is nested in the *JINITransportAgent* class, and it might access all of its data fields and methods. Hence, the methods, exposed by the *IJTABackendProtocol* interface and implemented by the *JTABackend* class, perform their server-side execution by using the *JINITransportAgent*'s attributes and methods.

Class *JINITransportAgentProxy*. This class plays the role of the JTA proxy. It implements the *Serializable* interface, which is a requirement coming from the JTA

JINI nature [3, 7]. The *Serializable* interface assures that the proxy can be saved to a byte stream and sent to the LUS, and then sent to the GIPSY execution node, requiring TA assistance. In addition, this class implements the *IJINITransportAgent* interface, which is the public interface used by the DGs and workers to communicate with the JTA.

The *JINITransportAgentProxy* implements a data filed *taBackend* that keeps a reference to the back-end object. This reference is received as a parameter by the class' constructor. The proxy uses this reference to communicate with the back-end. The following listing depicts two of the methods implemented by this class.

```
private JTABackendProtocol taBackend;
....
public Serializable getDemand(Uuid idDemand) {
        try {
                IWorkDemand oDemand;
                oDemand = taBackend.fetchDemand(idDemand, sClientIPAddr);

                return oDemand;
        }
        catch (Exception e) {
                printOutError(e.getMessage());
                return null;
        }
}

public Serializable getResult(Uuid idResult) {
        try {
                IWorkResult oResult;
                oResult = taBackend.fetchResult(idResult, sClientIPAddr);

                return oResult;
        }
        catch (Exception e) {
                printOutError(e.getMessage());
                return null;
        }
}
```

Listing 4.13. Coding *JINITransportAgentProxy* Class – Partial Implementation

The methods depicted above use the *taBackend* data field to execute the server-side *JTABackend* implementation.

4.3. Possible Implementations

In this section, we provide some guidelines for possible implementation of some of the DMS' components. In addition, some sample code is provided with a brief description.

4.3.1. Dispatcher Proxy with Distributed Events

In our current DMS implementation, the Dispatcher Proxies (DPs) perform constant reading in order to get the desired demands. Since the *DispatcherProxy* class instances run in a separate thread of control (see Fig.3.16), this does not overload the DP implementer – a TA, DG or worker. This solution is quite efficient due to the fact that the DP is not designed to perform a parallel work (see section 3.4.2). In the course of this thesis, we investigated different possible forms of parallelism for our DP. As a result of this investigation, we found a solution to the problem, based on distributed events.

In our current DMS implementation, the Demand Space (DS) is based on JavaSpace, which is a JINI implementation (see section 4.1.1.4). We could possibly implement the JavaSpace event-driven model that uses the distributed event model of JINI. This model delivers remote events to remote listeners. Hence, our DS will act as a source of events, where each entry written to the DS raises an event, notifying the DS clients for the newly received entry. JavaSpace provides a mechanism that allows the clients to register for events, raised by entries that match a specific template [7]. Those clients should implement the *RemoteEventListener* API interface, i.e. the DPs should implement that interface. The interface exposes a method called *notify()*, which is invoked by JavaSpace in order to send a remote event.

Therefore, in our possible implementation the DS will compare each entry written to the space with a template associated with a registration. This helps the DS to determine whether a *RemoteEventListener,* implemented by a DP, should be notified of the existence of a new entry that matches the template.

We propose an inner class called *DemandListener* that will implement the *RemoteEventListener* interface. This class will be nested in the class *DemandDispatcher.* The last will register the former for remote events. This technique allows the *DemandListener* class to call internally the appropriate *DemadDispatcher'* methods when a demand is found. In general, there is no reason that the class that registers for events cannot be the same class that receives them. However, creating a separate inner class for the event listener results in a clearer design.

Listing 4.14 depicts a possible implementation of the *DemandListener* class. The class' constructor uses the API class *UnicastRemoteObject* to export the

DemandListener class instance to the RMI runtime [3, 21]. When JavaSpace raises an event to signal the arrival of a matching entry, the space using the RMI runtime will invoke the *notify()* method of the *DemandListener*.

```
public  DispatcherProxy {
....
private class DemandListener  implements RemoteEventListener {

        private Uuid theID;
        private DemandState theState;

        public DemandListener (DemandState theState, Uuid theID) {

                this.theState = theState;
                this.theID = theID;

                UnicastRemoteObject.exportObject(this);
        }

        public void notify(RemoteEvent event) {

                if (theState.isPending())
                        readDemand();
                else
                        readResult(theID);

        }
....
}
```

Listing 4.14. Coding *DemandListener* Class

The class accepts two parameters - demand state and unique id. Those parameters are saved in local data fields, and used later by the *notify()* method. This method calls the *readDemand()* or *readResult()* *DispatcherProxy*'s methods, depending on whether the listener listens for demands or results. When the *readDemand()* and *readResult()* methods are called by the *notify()* method, they will be not blocked for reading, because there is an entry in the DS matching the template and the *read* operation will be performed immediately. A possible implementation of the *DemandListener* will result in modifications on the *DispatcherProxy* class.

4.3.2. Peer-to-Peer Transport Agent

In the course of the JINI investigation, we investigated another Java technology called JXTA [30]. JXTA is a framework that implements the peer-to-peer (P2P) communication model. In this model, the communication nodes have the same capabilities and each one can initiate a communication session. Some peer-to-peer

communications are implemented by giving each communication node both server and client capabilities [30].

Our possible P2P TA implementation will benefit from higher scalability, performance and simplicity. P2P has been demonstrated to support as many simultaneous users as the largest centralized systems [30]. Whereas, our current JTA implementation performs well with demands of size up to 18 MB (see section 5.3.2), a possible P2P TA will be able to handle efficiently demands with larger size.

Before going over some possible P2P TA implementation aspects, we describe the main JXTA concepts. The understanding of those concepts is critical for understanding the P2P TA implementation.

Peer. A *peer* is a P2P communication point, i.e. it is a virtual representation of a communication node. JXTA associates *peers* with special network services that they provide. Usually one *peer* resides on a single machine.

Endpoint. JXTA uses the *endpoints* to address the *peers*. It is the basic addressing method used by the JXTA applications to communicate with each other. An *endpoint* implements a specific protocol of communication. A peer can have many *endpoints*. Hence, a peer can implement different communication protocols, each being associated with an *endpoint*.

Pipe. JXTA uses the *pipes* as a virtual connection between peers. A *pipe* is an abstract layer on top of the communication protocols connecting two peers. Sometimes the communication between two peers is not direct. Those peers are connected via a gateway peers. The *pipe* makes the connection transparent, i.e. from the peer's perspective it is always straight.

The JXTA programming model has similarities with this exposed by JINI [30]. Hence, the possible P2P TA implementation will be similar to the JTA implementation with respect to the JXTA architecture. For a complete P2P TA implementation picture, we need more detailed analysis on JXTA, which is part of our future work.

4.4. Summary

In this chapter, we have presented the implementation of our design solution as software. Whereas, our DMS design is generic and platform independent (except the JINI Transport Agent design), our implementation is highly influenced by the distributed middleware technologies JINI, CORBA, DCOM and .Net Remoting. We have described some of the important aspects of these technologies in order to introduce the reader to the DMS implementation. The following elements describe the most important aspects of this implementation.

- For our implementation, we adopted the Java programming language. This is the same language used for implementing the other components of the GIPSY.
- In the course of this thesis, we implemented only the JINI Transport Agent (JTA). The transport agents based on CORBA, DCOM and .Net Remoting are part of our future work.
- For the implementation of our Demand Space (DS), we relied on JavaSpace.
- The JTA is built on top of JINI and RMI.
- We investigated some possible implementations like event-driven Dispatcher Proxy and Peer-to-peer Transport Agent, these being possible extensions of our current implementation.

Chapter 5: Experimental Results

> *If you want and expect a program to work, you will be more likely to see a working program—you will miss failures.*
>
> Cem Kaner et al.

This chapter contains selected test examples that address important issues like heterogeneity, parallelism, scalability, reliability and very high accuracy of our DMS. Wherever possible the experimental results obtained in the course of this testing are presented in a comparative format. The chapter starts with the testing environment, and goes over the test applications and obtained results.

5.1. Testing Environment

We performed all the tests on eight machines with an operating system varying from Linux to Windows. JINI ver. 2.0.1 and Java[TM] 2 Platform, Enterprise Edition 1.4 SDK were installed on all machines.

The following table represents the characteristics of the testing machines:

CPU	Speed	RAM	Operating System
PC Pentium II	333 MHz	128 MB	Windows 2000 Server.
PC Pentium III	650 MHz	256 MB	LINUX Red Hat 8.0
PC Pentium IV	3.01 GHz	512 MB	Windows XP Home Edition
PC Pentium IV	2.80 GHz	1 GB	Windows XP Professional Edition
PC Pentium IV	2.80 GHz	1 GB	Windows XP Professional Edition
PC Pentium IV	2.80 GHz	1 GB	Windows XP Professional Edition
PC Pentium IV	2.80 GHz	1 GB	Windows XP Professional Edition
PC Pentium IV	2.80 GHz	1 GB	Windows XP Professional Edition

Table 5.1. Testing Machines

We chose machines from three generations, which demonstrates the low resource needs required by the DMS.

5.2. JTA Worker

For our testing we created a prototype of the JTA worker called WorkerJTA. This application runs as a standalone application and relies on the JTA for accessing the DD. The WorkerJTA behaves like any worker. It gets pending demands from the DD, computes them, and returns back to the DD their computed result. In all the tests, we use the same worker, deployed on all the test machines. Right after its start, the worker connects with the JTA and starts listening for pending demands.

Listing 5.1 depicts a code segment implementing demand processing.

```
DispatcherEntry oEntry;
IWorkDemand oDemand;
IWorkResult oResult;

((IJINITransportAgent) oJTA).setClientIPAddress(sIPLocalAddress);

while (true) {
        oEntry = (DispatcherEntry)(((IJINITransportAgent) oJTA).getDemand());
        oDemand = (IWorkDemand)(oEntry.oObject);
        printOut("Demand received: name: " + oDemand.sName + " , id: " + oEntry.oUniqueID);
        oResult = oDemand.Execute();
        printOut("Demand computed");
        ((IJINITransportAgent) oJTA).setResult(oResult, oEntry.oUniqueID);
        printOut("Result dispatched");
}
```

Listing 5.1. Coding "Processing a Demand"

The oJTA variable is an object holding a reference to the JTA. The WorkerJTA uses that reference to execute the *getDemand()* method in order to get a pending demand, and the *setResult()* method in order to send a computed demand. Each demand has an *Execute()* method that computes the demand and generates its result.

5.3. Test Application "Remote Screenshot"

With this application, we generate demands that in their computation process take a screenshot of the host machine's desktop. The application consists of a DG for generating screenshot demands and demand implementation. The screenshot is moved to the DG as a result of the demand execution, then the DG saves the screenshot as a PNG file. The application uses Java's image I/O package introduced with version 1.4 to write the image file and Java's *java.awt.Robot* class to capture the desktop's screenshot.

Listing 5.2 depicts a code segment implementing the "Take a screenshot" procedure.

```
Toolkit theToolkit = Toolkit.getDefaultToolkit();
Dimension screenSize = theToolkit.getScreenSize();
Rectangle screenRect = new Rectangle(screenSize);

//**** create screen shot
Robot theRobot = new Robot();
theImage = theRobot.createScreenCapture(screenRect);

theScreenShot = new SerializableImage(theImage);
```

Listing 5.2. Coding "Take a Screenshot"

The code above determines the resolution of the screen in pixels and copies that information to a *Rectangle* object [23]. Then a *Robot* object is created and a screen shot is made using the *Rectangle* object. Note that we convert the image to a *SerializableImage*, i.e. we make the image *Serializable*, which is necessary in order to be transported via RMI. We created the *SerializableImage* class as a helper class for having the screenshot serialized.

5.3.1. Testing Heterogeneity

In this test, we run our workers on three different platforms – Windows XP, Windows 2000 and Linux Red Hat 8. The Windows XP machine hosts the DD and JTA. In addition, on this machine we run the DG generating screenshot demands.

The following elements depict the test case scenario:

On machine one:

- Run the Demand Space – JavaSpace.
- Run a JTA.
- Run a local DG (does not need a JTA to connect with the DD), generating screenshot demands.
- The DG generates three screenshot demands and sends them to the DD.
- Run a worker that connects with the JTA, gets a demand from the DD, computes the demand (takes a screenshot of the machine's desktop), and sends the result back to the DD via the JTA.
- The DG reads all the three results and dumps them into files.

On machine two:

- Run a worker that connects with the JTA on machine one, gets a demand from the DD, computes the demand (takes a screenshot of the machine's desktop), and sends the result back to the DD via the JTA.

On machine three:

- Run a worker that connects with the JTA on machine one, gets a demand from the DD, computes the demand (takes a screenshot of the machine's desktop), and sends the result back to the DD via the JTA.

The tests succeed in taking screenshots from different OS platforms like Linux, Windows 2000 and Windows XP. This demonstrates the heterogeneous nature of our DMS. The testing results can be seen in Appendix A.

5.3.2. Testing DMS Capacity of Big Demands Migration

In this test, we run one worker on a Windows 2000 machine and one DG on a Windows XP machine. The last hosts also the DD and JTA. The DG generates screenshot demands. These demands, at the time of their execution, clone the screenshot multiple times in order to increase the size of the *result* that must be delivered back to the DG. The DG estimates the process time in milliseconds and the size of the *delivered result* in Kb, and stores the data in a file. We performed seven test cases with an increasing size of the *delivered result*. The testing results are presented in a tabular as well as in a graphical manner.

The following table represents the testing results.

Test case	Result size Kb	Time (milliseconds)
Case 1	3087	26778
Case 2	6174	41319
Case 3	9261	60056
Case 4	12348	80576
Case 5	15435	101156
Case 6	18522	126312
Case 7	21609	error

Table 5.2. Results of "Testing DMS Capacity of Big Demands Migration"

Fig. 5.1 represents the testing results in a graphical manner. The numbers on X-axis represent the values of the message *(result)* size obtained during the different test cases. The Y-axis represents the time in milliseconds spent on processing a snapshot demand.

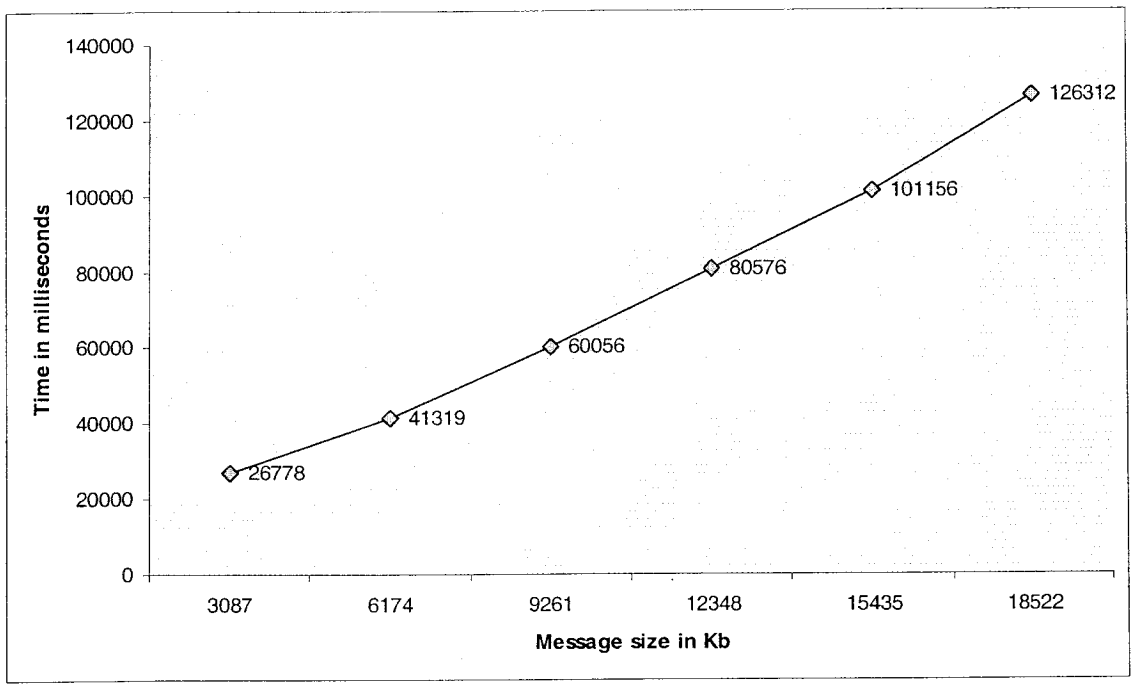


Fig.5.1. Results of "Testing DMS Capacity of Big Demands Migration"

Result analysis. In this test, we used the DMS to migrate huge computed demands (results) with a size varying up to 21609 Kb (approximately 20Mb). The migration was successful for all the demands sized up to 18522 Kb (approximately 18Mb), and unsuccessful for the demand with a size of 21609 Kb. In the last case, the migration was interrupted by an exception *java.lang.OutOfMemoryError*, which is thrown when the JVM cannot allocate an object due to out of memory. Therefore, the capacity of our DMS of migrating big demands is very high. The DMS is capable of migrating objects with a size up to 18 Mb, without any additional memory allocation, which is practically unlimited.

The curve shown in Fig. 5.1 depicts the time-size correlation in the migration of big demands. This curve goes as a straight line, which necessitates a conclusion that the migration time of big demands is proportional to the size, which reveals a correlation with the network speed.

5.4. Test Application "Remote Pi Calculation"

One of the most important questions in our research was about the effectiveness of our solution, which question being derived from the more empirical question about the effectiveness of the distributed systems at all. In the computational world, effectiveness is often associated with performance. Therefore, we had to prove the higher performance of the GIPSY, when the computational tasks are distributed for computation among multiple execution nodes, than when the computational tasks are computed on one single execution node. Computing lightweight tasks is not a good example for measuring this performance, since the time for their migration is an order of magnitude higher than the time for their execution. Hence, we needed a task model that is time-consuming and worthy to be migrated. In the course of this thesis, we found the computation of Pi number an effective task model for proving the effectiveness of our solution.

What is Pi? "Pi is a numerical constant that represents the ratio of a circle's circumference to its diameter on a flat plane surface. The value is the same regardless of the size of the circle" [39]. For most calculations, the value of Pi can be taken as 3.14159. Pi can be computed to an infinite number of decimal places, which makes it one of the few concepts in mathematics whose mention evokes a response of recognition and interest in those not concerned professionally with the subject. It has been a part of human culture and the educated imagination for more than twenty five hundred years.

Computing Pi up to 5000 decimal places. There are many formulas for Pi computation. With the test application called "Remote Pi Calculation", we compute Pi to 5000 decimal places by applying the formula proposed by John Machin (1706) [43], who manipulated identities based on a series for the inverse tangent function (ATN) to arrive at:

$$Pi = 16*ATN(1/5) - 4*ATN(1/239)$$

, and combining this identity with the series definition for ATN:

$$ATN(x) = x - x^3/3 + x^5/5 - x^7/7....$$

The application called "Remote Pi Calculation" generates computational demands that compute Pi to 5000 decimal places (see Appendix B). The application consists of a DG for generating "Pi calculation" demands and demand implementation. The DG is able to calculate the demands locally or remotely. When the last takes place, the DMS is used to deliver those demands to JTA workers, which compute those demands remotely, and to deliver the computational result back to the DG. The DG generates different number of demands to be computed, and calculate the overall computation time.

5.4.1. Testing Effectiveness

In this test, we run five workers on five different Windows XP machines and one DG on another Windows XP machine. The last hosts also the DD and JTA. All the machines had equal computational resources (see Table 5.1). We performed five test cases, where each test case includes local and remote computation of Pi. The number of computations increases by one in each test case. Therefore, in the first test case, we computed Pi one time (locally and remotely), in the second test case we computed Pi two times (locally and remotely), and so on. In all the test cases, we ensured the highest possible parallel computation of the remote demands. In each test case, different JTA workers run on different machines did the remote computations in parallel, and the DG did the local computations on another machine sequentially.

The following table represents the testing results.

Number of Pi calculations	Local	Remote
One time	28762	52476
Two times	82138	52365
Three times	136116	53654
Four times	205806	54123
Five times	261446	55712

Table 5.3. Testing Results of "Computing Pi"

Fig. 5.2 represents the testing results in a graphical manner. The X-axis represents the number of Pi computations, each computation being associated with a "Pi calculation" demand. The Y-axis represents the time in milliseconds spent on

processing all the "Pi calculation" demands. Fig. 5.2 depicts two curves – one for the local Pi computations and one for the remote Pi computations.

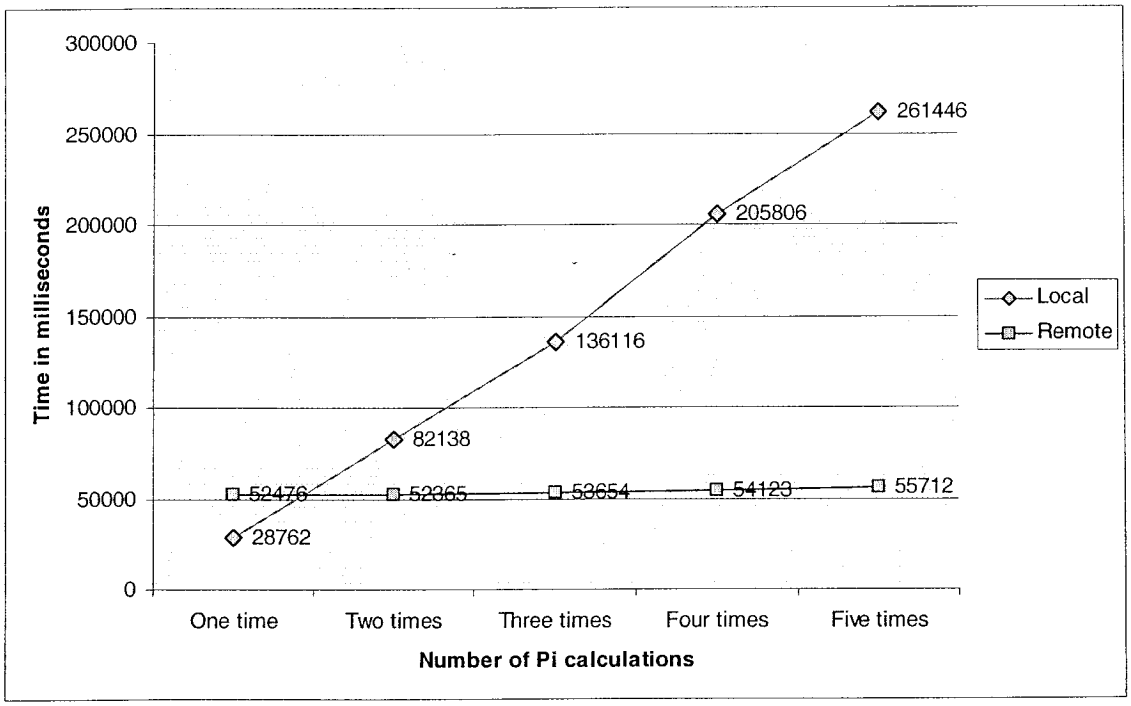


Fig.5.2. Testing Results of "Computing Pi"

Result analysis. In this testing, we simulated the ideal environment, where the execution nodes do nothing but the computation of demands and for each remote demand, there is a single machine. In addition, all the machines have the same computational resources. Hence, the testing results represent a pure performance (effectiveness) estimation of our solution.

The curve depicting the local Pi computations (see Fig.5.2) goes as a straight line crossing the zero point at an angle of 45 degrees. This line demonstrates a strong correlation between the computation time and number of Pi calculations, i.e. the computation time increases proportionally to the number of computations. This is a corollary to the sequential computation of the local demands by the DG.

The curve depicting the remote Pi computations (see Fig. 5.2) goes also as a straight line, but parallel to the X-axis. Hence, the computation time for all the remote demands is a constant and does not depend on the number of demands (i.e. number of Pi computations). This is a corollary to the parallel computation of the remote demands by the JTA workers, each being run on a different machine.

In Fig.5.2, we observe that the remote computations of Pi are much more effective than the local ones. In the first test case, the remote computation takes more time than the local one, which is due to the time needed by the DMS for migrating the demand to an execution node, and its corresponding result back to the DG. In all the consecutive test cases, the local computations take much more time than the remote ones. Since in the testing the number of machines is always sufficient for having all the remote demands executed in parallel, the efficiency depends mainly on the network performance.

The testing results reveal that the DMS used as a distributed middleware ensures very high effectiveness in case we have heavy time-consuming demands, and this effectiveness depends mainly on the number of machines hosting the JTA workers and the network performance.

5.5. Test Application "Hundreds Demands"

With this application, we generate up to 10000 lightweight demands (1Kb size) that simply generate a name consisting of a unique number and string. The application consists of a DG for generating demands and demand implementation. The DG calculates the overall process time for all the demands, and stores all the results of the computed demands in a file (see Appendix C). With this application we aimed at stress testing, which had to show how the DMS deals with a big amount of demands.

5.5.1. Testing Hot-plugging

In this test, we run all the DMS' components on one Linux machine. With the test, we aimed at the *hot-plugging* capability of our DMS. The following elements depict the test case scenario:

- Run the Demand Space – JavaSpace.
- Run a JTA.
- Run a JTA worker.
- Run a DG "Hundred Demands" generating 10000 demands.
- The JTA worker connects with the JTA, and starts getting and executing demands and generating results.
- Run another JTA worker.

- The second JTA worker connects with the JTA, and starts getting and executing demands and generating results.
- Stop the first worker, the second worker continues to work as usual.
- Run second JTA.
- Run another worker that connects with the second JTA.
- The third worker starts getting and executing demands and generating results.
- Stop the first JTA.
- The second worker stops automatically, after detecting the absence of the first JTA.
- The third worker continues to work.
- Meanwhile, the DG generates demands and receives results without being affected by the *hot-plugging* and shut down of the DMS' components.

In this test, we defined five DMS fictive states, determined by the active DMS' components. The DMS transits from a state to a state when some inactive DMS' components plug-in or some active DMS' components plug-out the system.

The following table represents the testing results in a tabular manner.

DMS states	Active DMS' component yes/no					
	DG	JTA 1	Worker 1	Worker 2	JTA 2	Worker 3
State 1	yes	yes	yes	no	no	no
State 2	yes	yes	yes	yes	no	no
State 3	yes	yes	no	yes	no	no
State 4	yes	yes	no	yes	yes	yes
State 5	yes	no	no	no	yes	yes

Table 5.4. Results of "Testing DMS Hot-plugging"

Fig. 5.1 represents the testing results in a graphical manner. The X-axis represents the active DMS' components, each being associated with a different pattern. The Y-axis represents the different states of the DMS.

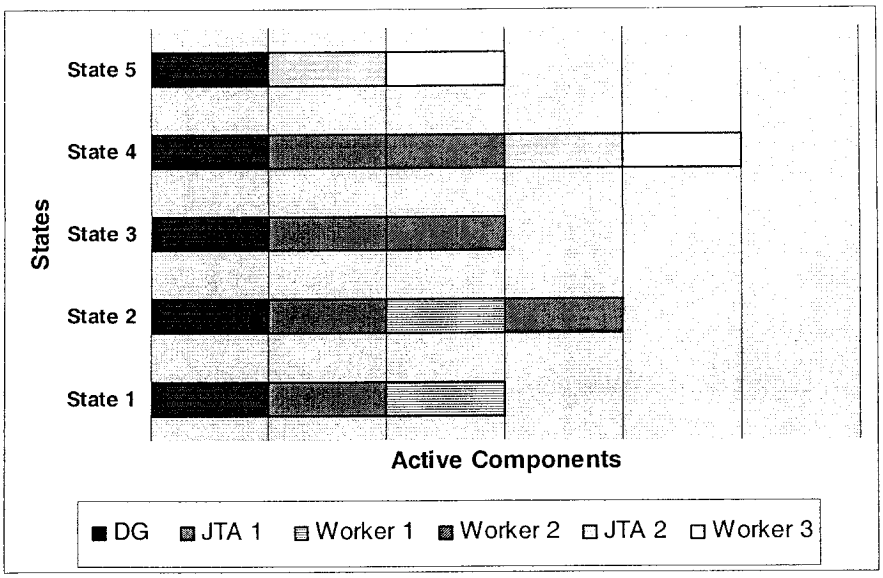


Fig.5.3. Results of "Testing DMS Hot-plugging"

In this test, it is assumed that the workers and DG are components of the DMS, since both run independently and plug-in and plug-out the DMS.

Result analysis. The testing results demonstrate a well-integrated *hot-plugging* system, allowing different components to go in and out without affecting the work of others.

5.5.2. Stress Testing

In this test, the DG generates up to 1000 lightweight demands (1Kb size). Here, we run three JTA workers, each one run on a separate Windows XP machine. In addition, we run a JTA and DD on another Linux machine. With this test, we aimed at software qualities like thoroughness, scalability, reliability and accuracy.

We specified ten test cases, each one with an increasing number of demands, starting from 100. In each test case, we calculated the process time for all the demands and the average process time per demand. The testing results are presented in a tabular as well as in a graphical manner.

The following table represents the testing results:

Test case	Number of demands	Process Time (milliseconds)	Average time (milliseconds)
Case 1	100	2233	22.33
Case 2	200	4056	20.28
Case 3	300	6519	21.73
Case 4	400	7821	19.55
Case 5	500	13730	27.46
Case 6	600	29302	48.84
Case 7	700	35221	50.32
Case 8	800	42190	52.74
Case 9	900	54829	60.92
Case 10	1000	82609	82.61

Table 5.5. Results of "Stress Testing"

Fig. 5.4 represents the correlation between the overall process time and the number of demands. The X-axis represents the test cases, each being represented by the number of processed demands. The Y-axis represents the overall process time in milliseconds.

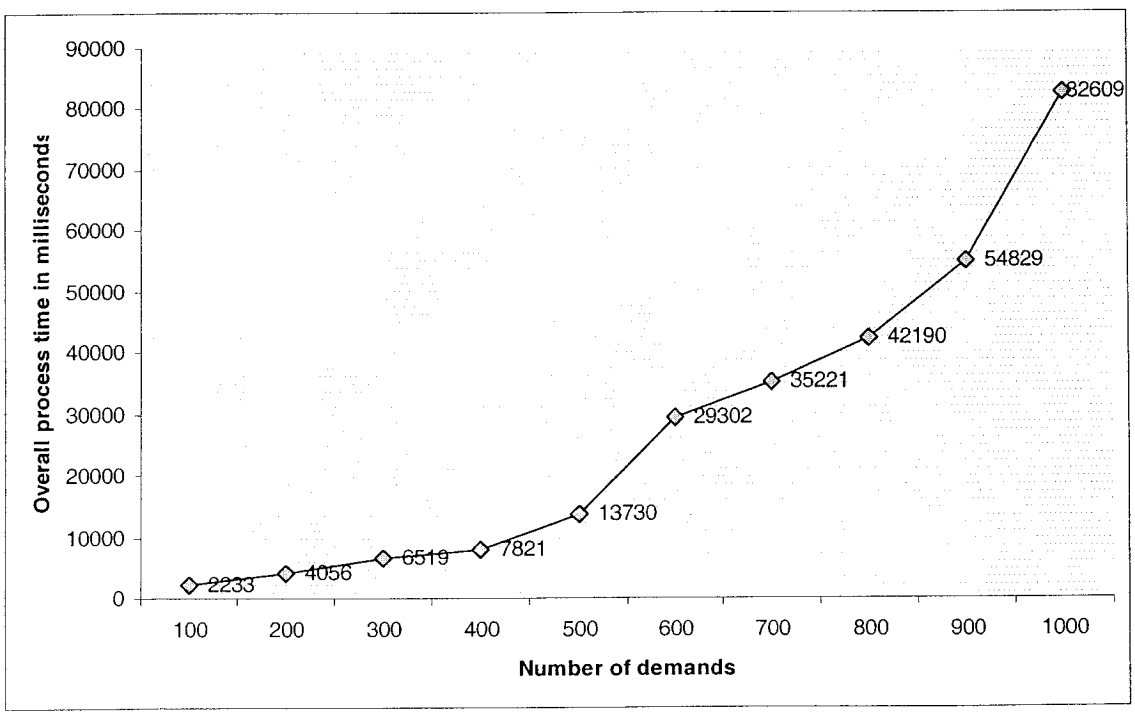


Fig.5.4. Correlation: Overall Process Time – Number of Demands

Fig. 5.5 is similar to Fig 5.4 but represents the correlation between the average process time per demand and the number of demands.

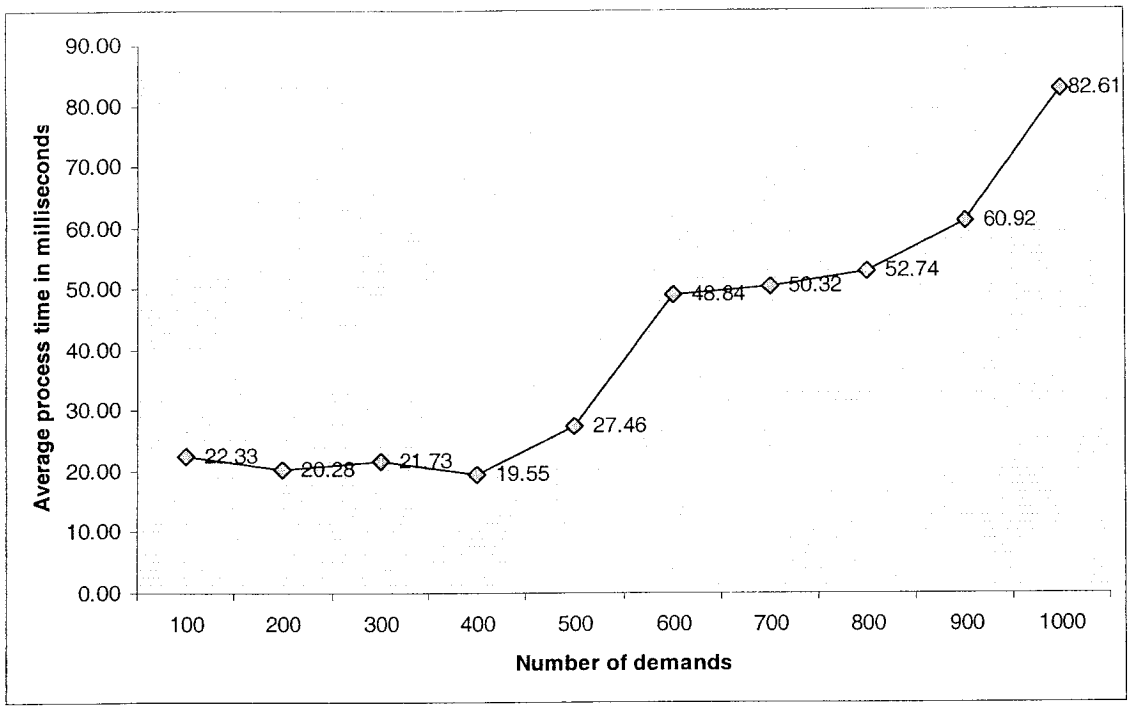


Fig.5.5. Correlation: Average Pocess Time – Number of Demands

Result analysis. The testing results demonstrate a system behavior proving high thoroughness, scalability, reliability and accuracy. From Fig. 5.5 we can see that up to 400 demands the average processing time for a single demand is around 20 milliseconds. Starting from 500 demands, the average processing time increases in a steady fashion to become 82 milliseconds for 1000 demands. This most likely is due to one of the following JavaSpace issues:

- The serialization overhead.
- The overhead of the network call.
- The scalability in terms of responsiveness / number of entries in the space
- The scalability in terms of responsiveness / number of concurrent users.

Another possible interpretation of these results is the resources of the DD machine, but it is less likely, since the machine is fast and with a sufficient resource amount. It is good to note that no demand was dispatched twice or lost, and no result was delivered to a wrong place. This demonstrates the high reliability and accuracy of our DMS. This test points to the conclusion that our DMS is highly scalable, but with the increase of workload, the average process time increases rapidly. Therefore, we observe an inverse dependency between the amount of workload and performance.

5.5.3. Hyper Stress Testing

In this test, we run two DG generates each generating 5000 lightweight demands (1 Kb size). Here also, we run eight JTA workers in pairs, each pair run on a separate Windows XP machine. In addition, we run four JTA, each being associated with two workers, and the DD on another Windows XP machine. With this test, we aimed at software qualities like thoroughness, scalability, reliability and accuracy.

Result analysis. All the demands were correctly dispatched, executed, and their result returned back to the DG. This demonstrates the very high accuracy and scalability of our DMS. The DMS migrated all the 10 000 demands forth and back in 1081446 milliseconds, which is 108.14 milliseconds average process time per demand. These times include the demand(s) execution time as well.

5.5.4. Performance Testing

In this test, we run one DG generating 300 lightweight demands (1Kb size) in each test case and different combinations of JTAs and JTA workers. The JTAs run on the same machine as the DG and DD, which is a Windows XP machine (see Table 5.1). The workers run as local (on the DD machine) or remote (on a Windows 2000 or Linux machine). In this test, we performed ten test cases, where the test cases differentiate in the combination of active JTAs and workers. With this test, we aimed at performance. Our goal was to prove that there is an inverse correlation between the overall process time and the number of JTAs and workers participating in the demand-computation process. The following table represents the test cases.

Test case	Number of Demands	Number of JTAs	Number of local workers	Number of workers on PIII	Number of workers on PII
Case 1	300	1	1		
Case 2	300	1		1	
Case 3	300	1	1	1	
Case 4	300	1	1	1	1
Case 5	300	1	2	1	1
Case 6	300	2	2	1	2
Case 7	300	2	2	2	2
Case 8	300	3	3	2	2
Case 9	300	3	3	3	2
Case 10	300	3	3	3	3

Table 5.6. Results of "Performance Testing"

Fig. 5.6 represents the testing results as a correlation between the process time and the number of JTAs and workers. The X-axis represents the test cases, each being associated with a specific set of JTAs and workers. The Y-axis represents the overall process time in milliseconds.

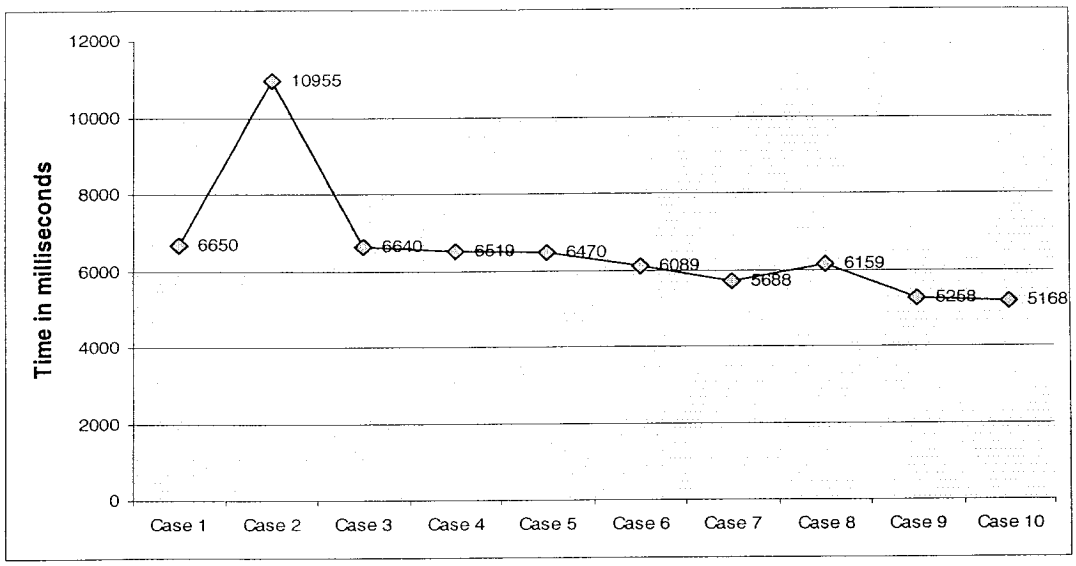


Fig.5.6. Results of "Performance Testing"

Result analysis. Fig.5.6 reveals two specific test cases – case #2 and #8:

- In case #2, the overall process time is much longer than the time in the other test cases.
- In case #8, we observe a slight increase of the process time, comparing to the decreasing time steady tendency in the other neighbour test cases.

The difference between case #1 and case #2 is that in case #2 the worker is remote one, but in case #1 the worker is local one. Therefore, the process time in case #2 is a sum of the process time estimated in case #1 and the network time. The network time is the time for moving the demands forward and backward through the network connecting the execution nodes.

Comparing case #1 and case #2, the performance is negative. However, in case #3, when the number of workers is two, we observe an increase of the performance - the process time goes down to the level estimated in case #1. This is in consequence of the parallel computation of demands, performed by the two workers. This makes up for lost network time.

Starting from case #3, we observe a steady tendency of decreasing the demand process time. Hence, we could conclude that the performance has a steady raise by increasing the number of workers and JTAs. An exception is case #8, which demonstrates an increase of the time due to the additional run of two processes – the JTA and worker, on the DD machine. These processes take CPU time and slow down the overall DD machine performance.

5.6. Summary

The experimental results confirm the high quality of our solution. By performing different tests, we have demonstrated that the implementation strictly follows the design and our DMS successfully fulfils the requirements stated in section 2.1. All the results are presented in a tabular and graphical format with given analysis.

The heterogeneity experiments demonstrate that our DMS is able to connect execution nodes run on different platforms.

The capacity experiments demonstrate that the current DMS implementation is able to transport demands with a size up to 18 Mb. For demands with a larger size, we need additional JVM memory extension.

The *hot-plugging* experiments demonstrate the high flexibility of our solution in terms of low coupling. All the DMS' components are "volunteers" that simply plug in and out the DMS at any time, without affecting the reliability of the system and the work of others.

The stress experiments demonstrate that the DMS is able to process thousands of demands with very high accuracy and reliability. Those experiments demonstrate also that our DMS is very scalable. The DMS is capable to accept an increasing workload, but with decreasing performance. The average processing time per demand increases rapidly with the increase of workload.

The effectiveness experiments demonstrate that the DMS, as a distributed middleware incorporated in the GIPSY ensures high effectiveness in case of heavy time-consuming demands. This effectiveness is in proportion to the time needed for the demand computation.

Finally, the performance experiments demonstrate direct proportionality between the number of active JTAs and workers and performance.

Chapter 6: Related Work

In theory, there is no difference between theory and practice.
But, in practice, there is.
Jan L.A. van de Snepscheut

In this chapter, we review related to our research work. The following elements describe some research projects pursuing similar to demand (object) migration themes.

6.1. Selected Scientific Projects

GLU. GLU [46] is a high-level system for parallel programming. GLU is a multi-language programming environment that relies on C and FORTRAN for the specification of its computational functions. In addition, Lucid [47] is used implicitly for expressing the parallelism of computational functions. GLU implements the eduction model of computation [47]. The basic eductive implementation architecture of GLU is called Centralized Generator Architecture (CGA). The CGA consists of a single generator and several workers, which architecture is very similar to the GIPSY architecture. The generator is able to propagate demands that must be executed by the workers. Like the GIPSY and DMS, the CGA incorporate *hot plugging* in terms of dynamic addition and deletion of workers.

DOME. DOME [48, 49] is an object based parallel programming environment for heterogeneous distributed networks. DOME exposes a library of distributed objects for parallel programming, where the programs are automatically distributed over a heterogeneous network. The DOME relies on PVM [50] for its process control and communication. Hence, the parallel computation is possible across the nodes of the current PVM machine. In order to achieve a parallel execution, DOME distributes the code and data among the PVM nodes.

Erlang. Erlang [51] is a concurrent functional language also used for distributed applications. Although its distribution mechanism is based on traditional concepts, its model does not need an interface description language (IDL). Distributed Erlang is used in implementing large software projects within the Ericsson group.

Mozart. Peter Van Roy et al. talk in [52] about a reliable object migration protocol that is part of the implementation of the Mozart [53] programming system. Mozart is an advanced development platform for intelligent, distributed applications. Mozart is base on the Oz language [54]. Oz is a concurrent object-oriented language with dataflow synchronization. The object migration protocol described in [52] is a centralized object migration mechanism that is able to "predict the network behavior" [52] with a fault tolerance policy. In this protocol, each object has a "home site", to which all migration requests are directed. The protocol defines "freely mobile" objects that have locally executable methods, i.e. each method executes in the thread that invokes it. The protocol works with so-called *distribution graphs* [52]. A distribution graph consists of active entities called nodes. The nodes have an internal state that can send and receive messages. All graph transformations are atomic and initiated by the nodes. The protocol uses a global unique name scheme to identify uniquely the nodes.

6.2. Commercial Projects

Reptile. Reptile [55] is a content exchange mechanism, based on peer-to-peer [30] and Web technologies. Reptile enables users to manage information in a distributed uniform environment. Reptile provides a distributed and secure mechanism for finding, accessing, and selectively sharing information.

VistaRepository. VistaRepository [56] provides distributed or centralized storage and scalable, flexible, secure, and unified access to resources (objects of different kinds), by exposing a peer-to-peer architecture (based on JXTA [30] technology). VistaRepository supports different storage types like file systems and RDBMS, by relying on framework that allows *hot plugging* of services. VistaRepository software is a Java program that works with a set of operational nodes.

Inferno. Inferno [57] is an operating system designed to be used in network environments. It provides interface to resources, scattered across the network, which allows the creation of distributed applications. Whereas in Inferno the interface to resources is specified at design time, the location of the code is assigned at run-time.

Chapter 7: Conclusion

A problem well stated is a problem half-solved.
Charles Kettering

In this chapter, we review our solution and draw conclusions.

7.1. Architecture

In the course of this thesis, we explored a process for engineering a software solution to the problem of demand migration in the GIPSY. In this thesis, we presented our generic approach to this problem and proposed a solution in the form of Demand Migration Framework (DMF).

Our DMF is a generic scheme for migrating objects in the form of functional demands, in a heterogeneous and distributed environment. The DMF has a layered architecture establishing a context for designing component-driven Demand Migration Systems (DMS) with loose coupling. The DMF layers fall into two major groups – Demand Dispatcher (DD) and Migration Layer (ML) (see Fig.2.1). The DMF relies on them to form architecture applicable to *asynchronous communication* systems, where the demands are delivered in a demand-driven manner, i.e. when they are asked. The DD incorporates a persistent storage mechanism called Demand Space (DS), for all the demands. The DS incorporates an Object Query Language (OQL) for querying the stored objects. A Presentation Layer (PL) brings the DS functionality to a more generic level. The ML establishes a high *fault-tolerant* and *secure* context for migrating objects from one node to another. It is designed in the form of messengers called Transport Agents (TA).

The DMF necessitates a design of standalone components that do not synchronize their data sharing, i.e. the DMF enables *hot-plugging* and *asynchronous communication*. By granting each demand with a GUID, the DMF assures *at least once-delivery semantics*.

The proposed DMF exposes a generic architecture with very high interoperability, which reveals a new communication model able to transport not only data but also real objects.

7.2. Design, Implementation and Results

In the course of this thesis, we applied the DMF to design, and succeeded to implement a DMS based on the distributed computing technologies JINI, CORBA, DCOM and .Net Remoting (see Fig.2.2). The DMS is a component driven system with loosely coupling – the components are highly independent. The DMS' components could stop, shutdown or start without affecting any other GIPSY' artifacts, which fulfills the requirement for *hot-plugging*.

The DMS inherits the DMF' layered structure with the two major groups of components - DD and TAs. They both work together to form the DMS as a demands propagator. The DD exposes for dispatching all the demands. The TAs contact the DD in order to deliver demands from Demand Generators (DGs), get and deliver those demands to workers, and deliver results, being associated with those demands, back to the DD and DGs (see Fig.2.3). In our design solution, the DD incorporates two components – Dispatcher Proxy (DP) and DS. Both are inherited from the DMF, and the DP is the design solution for the PL.

The DD distinguishes the demands by their state – *in process*, *pending* and *computed*. The TAs and workers work with a copy of an *in process* demand. The DD keeps the original demand in the DS as long as its result (*computed* demand) is not received. This *fault-tolerance* policy, assures that any demand will be re-dispatched, if an errorenous event occurs.

The TAs form a *multiplatform transport protocol*. They are based on different distributed technologies like JINI, DCOM, CORBA and .Net Remoting. A TA works independently for its own, with no concerns in the work of other TAs. The TAs are able to cope with the DD, DGs and workers and deliver demands and results to their recipients.

In the course of this thesis, we implemented a TA based on JINI. JINI is a Java based distributed technology and fits naturally to GIPSY, since the last is Java based as well. In our implementation, the DMS relies heavily on JINI, which is the core for two major DMS' contributors – the DS and JTA. The JTA is a JINI service, and any worker or DG that intends to use the JTA must adhere to its interface. For the implementation of the DS, we relied on JavaSpace. The last is a JINI implementation having all the characteristics of an object-oriented database.

With a DMS implying different distributed technologies we achieved *platform interoperability*, since our solution is able to connect GIPSY nodes spread among different platforms. In addition, the use of multiple distributed technologies, with the ability of adding new ones, turns our DMS into an *easy-upgradeable* system.

Since the DMS relies on the DS for keeping all the pending demands and results, it does not depend on the recipients – DGs and workers. They will receive their results and demands when they apply for them, which is an implication of *asynchronous messaging*. In addition, the DMS does not discriminate the workers and demands by importance. Hence, the DMS facilitates the workload distribution - any worker or DG will be served in a similar manner, when it is possible.

The DMS is well secured system. It adheres to the traditional Java security mechanism and distributed technology implemented security.

In the course of this thesis, we performed experimental testing, and demonstrated that our solution fulfils the requirements stated at the beginning of this thesis. The experimental results demonstrated the heterogeneity, *hot-plugging*, high scalability, high accuracy and reliability of our DMS. The experimental results reveal some problems, mostly related to performance and efficiency. Here we should note that our solution deals only with demand migration aspects and does not deal with load balancing and efficiency aspects. The last are to be tackled by other subsystems of the GIPSY and future subsystems of the DMS.

The DMS plays an important role in the GIPSY, but also it has a promising future as an independent communication system, able to transport data and behavior in heterogeneous environments.

Chapter 8: Future Work

> *Abstraction is one of the fundamental ways that we as humans cope with complexity.*
>
> Grady Booch

This section describes future work that will be performed as a direct consequence of the research undertaken in the course of this thesis.

8.1. Architecture

The proposed DMF/DMS architecture does not include any performance enhancements. With our future work on this architecture, we are aiming at performance and load balancing issues.

8.1.1. Dispatcher Supervisor

Our DMF does not incorporate any monitoring tools. We have an idea of implementing an Observation Layer (OL), which will have some functions for monitoring and control over the DD. Such a layer will tackle with load balancing and efficiency aspects of the demand dispatching. A possible OL component could be a garbage collector for all the old demands and results, which have been not dispatched for a certain amount of time.

8.1.2. DD Cache

In the computational world, the presence of a cache system enhances the overall performance. We have an idea of implementing a Demand Dispatcher Cache (DDC) layer that will be responsible for a great deal of the system performance improvement. The DDC will be an internal DD's subsystem that will stay on top of the DS, and that will act as a buffer for recently-used *computed* demands. This will improve the performance, since any subsequent demand having stored result in the DDC will be not processed for dispatching. Instead, the DD will simply return its computed result, stored in the DDC.

8.2. Design and Implementation

All the architectural enhancements will be designed and implemented. Meanwhile, we continue with the completion of our current architecture, which includes the following aspects of design and implementation.

8.2.1. Transport Agent Based on CORBA, DCOM and .Net Remoting

Concerning the TAs, we are going to design and implement TAs based on DCOM, .Net Remoting and CORBA. This work requires integration of these technologies with Java. The following elements describe some possible solution to the problem:

VisiBroker for Java. For the CORBA integration, we are planning to use a popular commercial ORB. In addition, we are going to use the IDL to define interfaces of our CORBA TA. A possible ORB is Borland VisiBroker for Java [31]. This ORB has been licensed by several different companies, including Oracle, Netscape, and Novell.

JACOB project. For the DCOM integration, we could use some kind of bridge between the COM objects and Java. A possible solution is the JACOB project [32]. This open-source project connects JAVA and COM and allows calling of COM automation components from Java.

Ja.NET. For the .Net Remoting integration, we could rely on Ja.NET, which is a "bridge between the world of Java and the world of Microsoft.NET" [33]. Ja.Net generates Java proxies that expose or consume components using the .Net Remoting protocol.

The applications mentioned above fall in the group of third-party software. This decreases the interoperability and independence of our solution. Fortunately, Java allows calling non-Java code, usually referred to as native code. Therefore, we could create our own integration library that will call the functions of the CORBA TA, DCOM TA and .NET Remoting TA, all being implemented in C++.

In addition to the TA implementations described above, we are planning to implement the Java Message Service (JMS) as a possible alternative for the JTA.

Another alternative for the JTA will be the P2P TA. This TA will be similar to the current JTA implementation, but implementing P2P communication protocol (see section 4.3.2).

8.2.2. Security Enhancements

Colouris in [17] talks in general about the security of distributed systems and covers in details encryption and key distribution, which we are planning to implement in our future DMS security model.

References

[1] J. Paquet, P.G. Kropf, "The GIPSY Architecture", *LNCS,* Vol. 1830, p. 144–153, 2000.

[2] M. Fukuda, L.F. Bic et al. "Messages versus messengers in distributed programming", *ICDCS*, p. 347, 1997.

[3] W. K. Edwards, "Core JINI", Upper Saddle River, NJ, Prentice Hall PTR, 1999.

[4] W. Emmerich, "Engineering Distributed Objects", Baffins Lane, Chichester, Wiley, 2000.

[5] J. Paquet, "Intensional Scientific Programming", PhD Thesis, Laval University, Quebec, Canada, 1999.

[6] J. Day, "The (un)revised OSI reference model", *ACM SIGCOMM Computer Communication Review*, vol. 25, p. 39–55, Oct 1995.

[7] R. Flenner, "Jini and JavaSpaces Application Development", Indianapolis, Sams Publishing, 2001.

[8] N. Carriero and D. Gelernter, "How to write parallel programs: A guide to the Perplexed, *ACM Computing Surveys*, vol. 21, no. 3, p. 323 – 357, Sept. 1989.

[9] C. Amza, A.L. Cox et al., "TradeMarks: Shared Memory Computing on networks of Workstations", *Computer*, vol. 29, no. 2, p. 18–28, Feb 1996.

[10] "Process migration", ACM Computing Surveys, vol. 32, no. 3, p. 241-299, Sept. 2000.

[11] K. Pingali and G. Arvind, "Efficient Demand-driven Evaluation. Part 1," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 2, p. 311-333, 1985.

[12] K. Pingali and G. Arvind, "Efficient demand-driven evaluation", part 2, *ACM Trans. Programming Language Syst.*, vol. 8, p. 109-139, Jan. 1986

[13] R. Orfali and D. Harkey, "Client/Server Programming with Java and CORBA", Wiley, 1998.

[14] R. Grimes and R. Grimes, "Professional DCOM Programming", Wrox Press Ltd., Birmingham, UK, 1997.

[15] R. G. G. Cattell and D. K. Barry, "The Object Database Standard: ODMG 3.0", Morgan Kaufmann, 2000.

[16] I. Rammer and M. Szpuszta, "Advanced .Net Remoting", Apress Publishers, ver. 2, 2004.

[17] G. F. Coulouris and J. Dollimore, "Distributed systems: concepts and design", Boston, MA, Addison-Wesley, 1994.

[18] Philippe Kruchten, "Architectural Blueprints—The "4+1" View Model of Software Architecture", Rational Software Corp., 1995.

[19] Craig Larman, "Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process", Prentice Hall PTR, 2001.

[20] E. Gamma et al, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley Professional, 1994.

[21] Jim Farley, "Java Distributed Computing", O'Reilly, 1998.

[22] M. Pistoia et al, "Enterprise Java (TM) Security: Building Secure J2EE(TM) Applications", Addison-Wesley Professional, 2004.

[23] "Java (TM) 2 SDK, Standard Edition Documentation", ver. 1.4.2, Sun Microsystems Inc., 2003.

[24] M. Hicks et al, "Transparent communication for distributed objects in Java", *In proceedings of the ACM 1999 conference on Java Grande*, p.160-170, San Francisco, California, June 1999.

[25] "Persistent State Service Specification", ver. 2, OMG, 2002.

[26] N. W. Cluts, "A "COM artist" shares the secret of creating ActiveX controls", MSDN Library, Microsoft Corp., 1997.

[27] S. F. Wilson, "MCSD Training Kit for Exam 70-100: Analyzing Requirements and Defining Solution Architectures", Microsoft Press, 1999.

[28] J. Richter, "Applied Microsoft .Net Framework Programming", Microsoft Press, 2002.

[29] Mark Allman, "An evaluation of XML-RPC", *ACM SIGMETRICS Performance Evaluation Review*, vol. 30, no. 4, p. 2 - 11, March 2003.

[30] D. Brookshier et al, "JXTA: Java P2P programming", Indianapolis, Sams Publishing, 2002.

[31] "Borland Enterprise Server 6.5 VisiBroker Edition technical overview", Borland Corp., 2004.

[32] Dan Adler, "The JACOB Project: A Java-COM Bridge", ver. 1.8, 2004.

[33] "Application Interoperability: Microsoft .Net and J2ee (Patterns & Practices Series)", Microsoft Press, 2004.

[34] "Jini Architectural Overview", Sun Microsystems, Inc, California, 1999.

[35] J. Paquet and J. Plaice, "Dimensions and functions as values", *In proceedings of the Eleventh International Symposium on Lucid and Intensional Programming*, Sun Microsystems, Palo Alto, California, May 1998

[36] J. Paquet and J. Plaice, "The semantics of dimensions as values", *In Intensional Programming II*, World Scientific, 1999

[37] B. Lu, P. Grogono and J. Paquet, "Distributed execution of intensional multidimensional programming languages", *In Parallel and Distributed Computing and Systems - PDCS 2003*, Marina Del Rey, California, 2003

[38] J. Plaice and J. Paquet. "Introduction to intensional programming", *In Intensional Programming I*, p. 1-14, World Scientific, Singapore, 1996

[39] "Pi", Whatis?com, IT encyclopedia and learning center, http://whatis.techtarget.com/

[40] J. Rumbaugh et al, "The Unified Modeling Language: Reference manual", Addison-Wesley, 2000

[41] Garry Nutt, "Operating Systems – A modern perspective", ver. 2, Addison-Wesley, 2002

[42] "CORBA Basics", Object Management Group Inc., 2005

[43] "Machin's Formula", MathWorld at Wolfarm Research, 2005, http://mathworld.wolfram.com/MachinsFormula.html

[44] R. Jagannathan and C. Dodd, "GLU Implementation Architectures for Heterogeneous Systems", *Ninth International Symposium on Languages for Intensional Programming (ISLIP'96)*, Tempe, Arizona, May 1996

[45] N. Lynch, "Distributed Algorithms", Morgan Kaufmann Publishers Inc., 1996

[46] R. Jagannathan and C. Dodd, "GLU Implementation Architectures for Heterogeneous Systems", *Ninth International Symposium on Languages for Intensional Programming (ISLIP)*, Tempe, Arizona, May 1996.

[47] E. A. Ashcroft, "Dataflow and eduction: Data-driven and demand-driven distributed computation.", In J. W. deBakker, W.P. deRoever, and G. Rozenberg, editors, *Current Trends in Concurrency*, number 224 in Lecture Notes on Computer Science, pages 1 -- 50. Springer-Verlag, 1986

[48] A. Beguelin et al., "Dome: Distributed object migration environment", *Technical Report CMU-CS-94-153*, Carnegie-Mellon University, May 1994

[49] J. N. C. Arabe et al., "Dome: Parallel programming in a heterogeneous multi-user environment.", *Technical Report CMU-CS-95-137*, School of Computer Science, Carnegie Mellon University, Pittsburgh, April 1995.

[50] A. Geist et al., "PVM: Parallel Virtual Machine A Users' Guide and Tutorial for Networked Parallel Computing.", MIT Press, 1994

[51] C. Wikstrom, "Distributed programming in Erlang.", *In PASCO'94 - First International Symposium on Parallel Symbolic Computation*, Linz, Sept. 1994

[52] Van Roy et al., "A lightweight reliable object migration protocol", *Lecture Notes in Computer Science*, vol. 1686. Springer Verlag, 1999

[53] Mozart project, http://www.mozart-oz.org

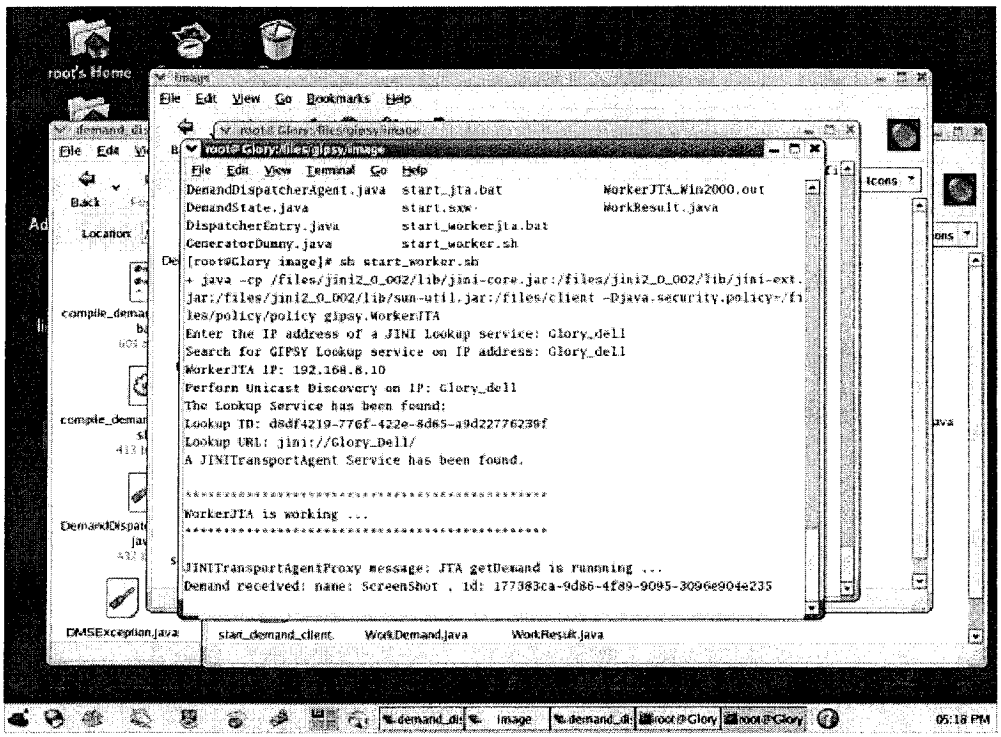
[54] Seif Haridi and Nils Franzen, "Tutorial of Oz", Technical report, Draft, *In Mozart documentation*, available at http://www.mozart-oz.org, 1999

[55] Reptile project, http://reptile.openprivacy.org

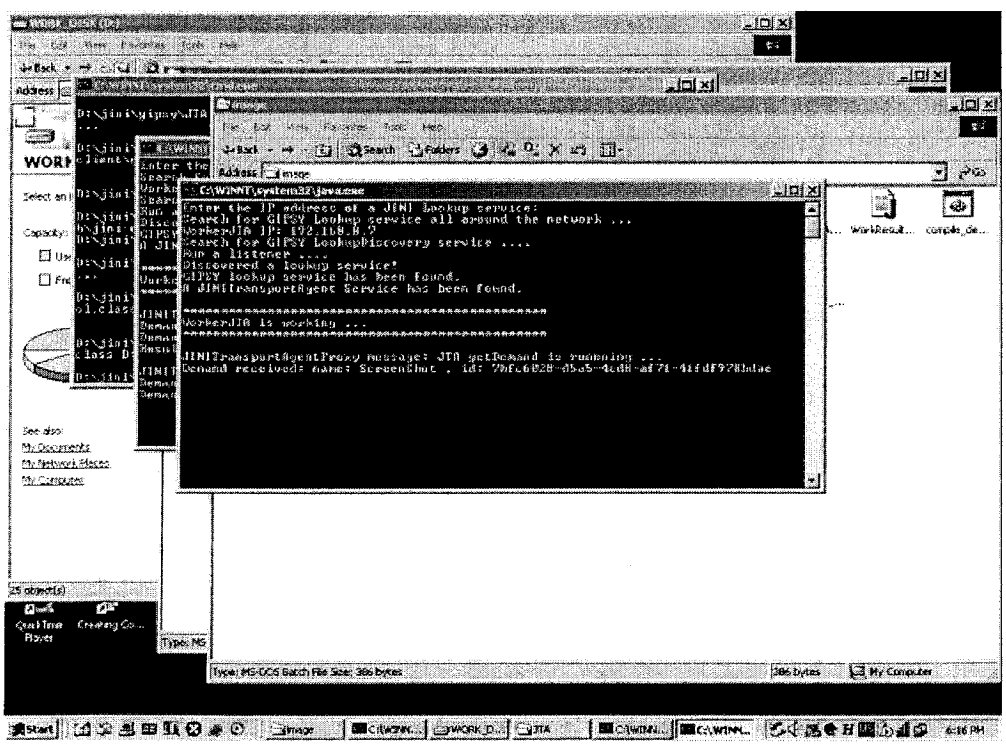
[56] VistaPortal Software inc., VistaRepository project, www.vistaportal.com/products/vistarepository.htm

[57] Lucent Technologies Inc., Inferno project, http://www.vitanuova.com/inferno

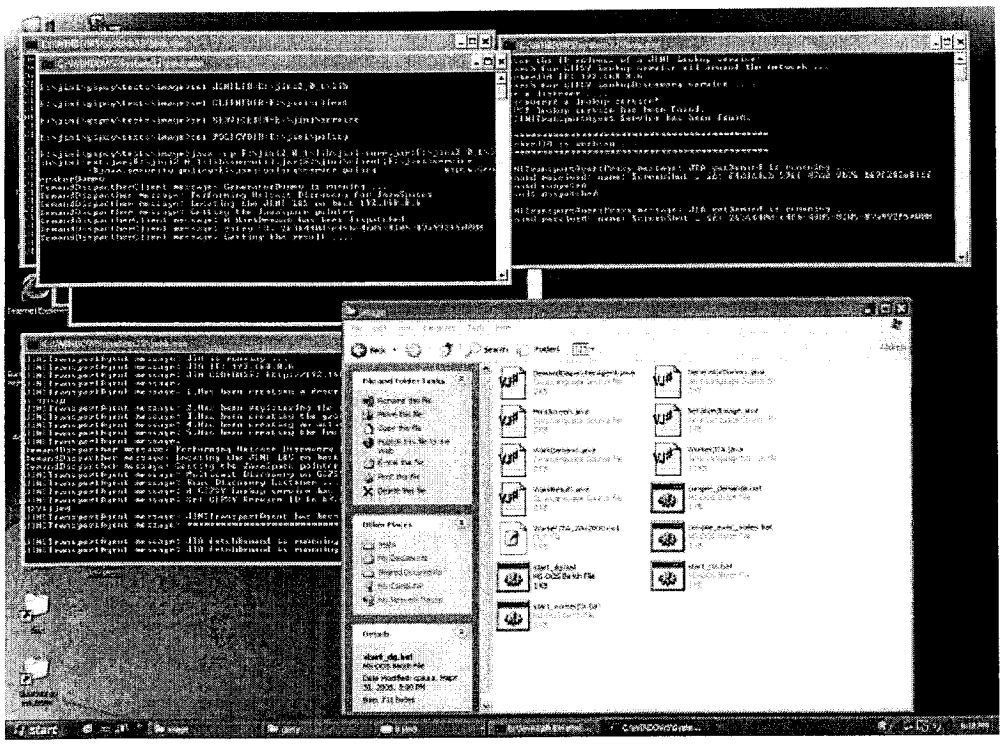
Appendix A: Results of Heterogeneity Testing



A screenshot taken from the Linux machine with IP: 192.168.8.10



A screenshot taken from the Windows 2000 machine with IP: 192.168.8.7



A screenshot taken from the Windows XP machine with IP: 192.168.8.6

Appendix B: Calculating Pi to 5000 Significant Digits

Start time: 1112720108735

192.168.8.5 , Pi calculation:
3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534211706798214808651328230664709384460955058223172535940812848111745028410270193852110555964462294895493038196442881097566593344612847564823378678316527120190914564856692346034861045432664821339360726024914127372458700660631558817488152092096282925409171536436789259036001133053054882046652138414695194151160943305727036575959195309218611738193261179310511854807446237996274956735188575272489122793818301194912983367336244065664308602139494639522473719070217986094370277053921717629317675238467481846766940513200056812714526356082778577134275778960917363717872146844090122495343014654958537105079227968925892354201995611212902196086403441815981362977477130996051870721134999999837297804995105973173281609631859502445945534690830264252230825334468503526193118817101000313783875288658753320838142061717766914730359825349042875546873115956286388235378759375195778185778053217122680661300192787661119590921642019893809525720106548586327886593615338182796823030195203530185296899577362259941389124972177528347913151557485724245415069595082953311686172785588907509838175463746493931925506040092770167113900984882401285836160356370760610471018194295559619894676783744944825537977472684710404753464620804668425906949129331367702898915210475216205696602405803815019351125338243003558764024749647326391419927260426992279678235478163600934172164121992458631503028618297455570674983850549458858692699569092721079750930295532116534498720275596023648066549911988183479775356636980742654252786255181841757467289097777279380008164706001614524919217321721472350141441973568548161361157352552133475741849468438523323907394143334547762416862518983569485562099219222184272550254256887671790494601653466804988627232791786085784383827967976681454100953883786360950680642251252051173929848960841284886269456042419652850222106611863067442786220391949450471237137869609563643719172874677646575739624138908658326459958133904780275900994657640789512694683983525957098258226205224894077267194782684826014769909026401363944374553050682034962524517493996514314298091906592509372169646151570985838741059788595977297549893016175392846813826868386894277415599185592524595395943104997252468084598727364469584865383673622262609912460805124388439045124413654976278079771569143599770012961608944169486855584840635342207222582848864815845602850601684273945226746767889525213852254995466672782398645659611635488623057745649803559363456817432411251507606947945109659604025228879710893145669136867228748940560101503308617928680920874760917824938589009714909675985261365549781893129784821682998948722658804857564014270477555132379641451523746234364542858447952658678210511413547357395231134271661021359695362314429524849371871101457654035902799344037420073105785390621983874478084784896833214457138687519435064302184531910484810053706146806749192781911979399520614196634287544406437451237181921799983910159195618146751426912397489409071864942319615679452080951465502252316038819301420937621378559566389377870830390697920773467221825625996615014215030680384477345492026054146659252014974428507325186660021324340881907104863317346496514539057962685610055081066587969981635747363840525714591028970641401109712062804390397595156771577004203378699360072305587631763594218731251471205329281918261861258673215791984148488291644706095752706957220917567116722910981690915280173506712748583222871835209353965725121083579151369882091444210067510334671103141267111369908658516398315019701651511685171437657618351556508849099898599823873455283316355076479185358932261854896321329330898570642046752590709154814165498594616371802709819943099244889575712828905923233260972997120844335732654893823911932597463667305836041428138830320382490375898524374417029132765618093773444030707469211201913020330380197621101100449293215160842444859637669838952286847831235526582131449576857262433441893039686426243410773226978028073189154411010446823252716201052652272111660396665573092547110557853763466820653109896526918620564769312570586356620185581007293606598764861179104533488503461136576867532494416680396265797877185560845529654126654085306143444318586769751456614068007002378776591344017127494704205622305389945613140711270004078547332699390814546646458807972708266830634328587856983052358089330657574067954571637752542021149557615814002501262285941302164715509792592309907965473761255176567513575178296664547791745011299614890304639947132962107340437518957359614589019389713111790429782856475032031986915140287080859904801094121472213179476477726224142548545403321571853061422881375850430633217518297986622371721591607716692547487389866549494501146540628433663937900397692656721463853067360965712091807638327166416274888800786925602902284721040317211860820419000422966171196377921337575114959501566049631862947265473642523081770367515906735023507283540567040386743513622224771589150495309844489333096340878076932599397805419341447377441842631298608099888687413260444
End time: 1112720161211

Duration: 52476

Appendix C: A Stress Test Record

Start time: 1112732098496

192.168.8.6 , Result_1
192.168.8.7 , Result_2
192.168.8.10 , Result_3
192.168.8.6 , Result_4
192.168.8.6 , Result_5
192.168.8.6 , Result_6
192.168.8.6 , Result_7
192.168.8.6 , Result_8
192.168.8.6 , Result_9
192.168.8.6 , Result_10
192.168.8.6 , Result_11
192.168.8.6 , Result_12
192.168.8.6 , Result_13
192.168.8.6 , Result_14
192.168.8.6 , Result_15
192.168.8.6 , Result_16
192.168.8.6 , Result_17
192.168.8.6 , Result_18
192.168.8.6 , Result_19
192.168.8.7 , Result_20
192.168.8.6 , Result_21
192.168.8.6 , Result_22
192.168.8.7 , Result_23
192.168.8.6 , Result_24
192.168.8.7 , Result_25
192.168.8.6 , Result_26
192.168.8.7 , Result_27
192.168.8.10 , Result_28
192.168.8.6 , Result_29
192.168.8.7 , Result_30
192.168.8.10 , Result_31
192.168.8.7 , Result_32
192.168.8.10 , Result_33
192.168.8.7 , Result_34
192.168.8.6 , Result_35
192.168.8.7 , Result_36
192.168.8.6 , Result_37
192.168.8.10 , Result_38
192.168.8.7 , Result_39
192.168.8.10 , Result_40
192.168.8.6 , Result_41
192.168.8.10 , Result_42
192.168.8.7 , Result_43
192.168.8.10 , Result_44
192.168.8.7 , Result_45
192.168.8.10 , Result_46
192.168.8.6 , Result_47
192.168.8.7 , Result_48
192.168.8.10 , Result_49
192.168.8.7 , Result_50
192.168.8.6 , Result_51
192.168.8.6 , Result_52
192.168.8.7 , Result_53
192.168.8.6 , Result_54
192.168.8.10 , Result_55
192.168.8.6 , Result_56

192.168.8.10 , Result_57
192.168.8.7 , Result_58
192.168.8.6 , Result_59
192.168.8.10 , Result_60
192.168.8.6 , Result_61
192.168.8.7 , Result_62
192.168.8.6 , Result_63
192.168.8.10 , Result_64
192.168.8.6 , Result_65
192.168.8.7 , Result_66
192.168.8.10 , Result_67
192.168.8.6 , Result_68
192.168.8.6 , Result_69
192.168.8.7 , Result_70
192.168.8.6 , Result_71
192.168.8.6 , Result_72
192.168.8.6 , Result_73
192.168.8.6 , Result_74
192.168.8.7 , Result_75
192.168.8.10 , Result_76
192.168.8.6 , Result_77
192.168.8.10 , Result_78
192.168.8.7 , Result_79
192.168.8.6 , Result_80
192.168.8.10 , Result_81
192.168.8.6 , Result_82
192.168.8.7 , Result_83
192.168.8.6 , Result_84
192.168.8.6 , Result_85
192.168.8.6 , Result_86
192.168.8.7 , Result_87
192.168.8.10 , Result_88
192.168.8.6 , Result_89
192.168.8.10 , Result_90
192.168.8.7 , Result_91
192.168.8.6 , Result_92
192.168.8.6 , Result_93
192.168.8.10 , Result_94
192.168.8.6 , Result_95
192.168.8.10 , Result_96
192.168.8.7 , Result_97
192.168.8.6 , Result_98
192.168.8.10 , Result_99
192.168.8.6 , Result_100

End time: 1112732100729

Duration: 2233

Index

J

L

M

N

O

P

R

S

T

U

W

X