IMPLEMENTATION AND COMPREHENSIVE STUDY OF DEMAND MIGRATION SYSTEMS IN GIPSY

Amir Hossein Pourteymour

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

Concordia University

Montréal, Québec, Canada

September 2008

Your file Votre référence
ISBN: 978-0-494-45710-8
Our file Notre référence
ISBN: 978-0-494-45710-8

Abstract

Implementation and Comprehensive Study of Demand Migration Systems in GIPSY

Amir Hossein Pourteymour

Intensional programming is a programming language paradigm based on the notion of declarative programming where the declarations are evaluated in an inherent multidimensional context space. Program identifiers are evaluated in a context, where each demand is generated, propagated, computed, and stored as an identifier-context pair.

General Intensional Programming System (GIPSY) is a hybrid multi-language programming system that overcame the limitation of previous Intensional Programming systems by designing a Demand Migration Framework (DMF) to provide a generic, dynamic, and technology-independent infrastructure.

A DMF instance, called a Demand Migration System (DMS), is used to propagate demands from one GIPSY execution node to another. A GIPSY program is executed using three components, each of which possibly having several instances, all of which possibly being executed on different nodes: the Demand Generator (DG), that generates demands according to the compiled Lucid program, the Demand Worker (DW), that executes procedure calls embedded in the Lucid program, and the DMS, that acts as a communication/storage middleware between the latter.

This thesis extends the previous investigations on the DMF by applying and extending DMF rationales and design to implement an instance of our DMS using Java

Message Service (JMS-DMS).

JMS-DMS is an investigation toward having the combination of two paradigms of Message-Oriented Middleware (MOM) and Event-Driven Architecture (EDA) to handle our demand-driven computation. We also investigate on the behavior of our instances in different perspectives such as latency, dispatching, availability, scalability, maintainability, and configurability, which complements our research toward having the robust Demand Migration System.

Acknowledgments

I would like to extend my gratitude first and foremost to my supervisor Dr. Joey Paquet as I am always indebted for his invaluable guidance and vision throughout this research. He believed in me from the first day and his encouragements and advices were my major inspiration for this work.

Words are inadequate to thank my parents, Lila and Ardeshir, and my lovely brother, Amirali, for their unconditional love, support, and immense source of inspiration for me all through my life. I should sincerely thank my lovely uncle and aunt, Amir and Nina, who endlessly and emotionally supported me in every single day of my life in Montreal.

I would also like to thank my friendly team members in the GIPSY lab who helped me from the first day specially Emil Vassev who as my encouraging mentor introduced me to his work and helped me to develop new ideas through this research with his insightful comments and suggestions. Many thanks go to Serguei Mokhov for his wonderful ideas and advices in our research.

This work has been sponsored by NSERC and the Faculty of Engineering and Computer Science of Concordia University, Montréal, Québec, Canada.

Last but not least, I would like to thank Nasim Farsinina for reviewing my thesis.

Contents

List of Figures

List of Tables

List of Abbreviations

DG	Demand Generator
DMF	Demand Migration Framework
DMS	Demand Migration System
DW	Demand Worker
EDA	Event-Driven Architecture
GEE	General Eduction Engine
GEER	General Eduction Engine Resource
GIPSY	General Intensional Programming System
GLU	DGranularLucid
IP	Intensional Programming
JINI-DMS	JINI - Demand Migration System
JMS	Java Message Services
JMS-DMS	JMS - Demand Migration System
MOM	Message-Oriented Middleware
QoS	Quality of Services
RMI	Remote Method Invocation
RPC	Remote Procedural Call
TA	Transport Agent

Chapter 1

Introduction

You were born with wings. Why prefer to crawl through life?

J.M.Rumi - Persian Poet (13th Century)

The General Intensional Programming System (GIPSY) is a hybrid multi-language programming system in a demand-driven execution environment [78]. GIPSY is aimed at the long-term investigation into the possibilities of *intensional* programming, more specifically the Lucid declarative and functional family of programming languages [56, 79, 38, 41, 22, 58]. GIPSY is an ambitious project directed by Dr. Joey Paquet in the department of Computer Science and Software Engineering in Concordia University in Montreal, Quebec.

This thesis focuses specifically on the architecture used for the distributed migration of the demands generated by the system at run time. From this perspective, the GIPSY evaluation engine, or General Eduction Engine (GEE) is a demand-driven execution system that is based on Demand Generators (DG) that determines the control process by generating demands and relies on Demand Worker (DW) to execute some of them [78]. Both DG and DW are the GIPSY execution nodes, and these nodes are potentially distributed, in case the demands are migrated via the network from the

generators to the workers and the results are flowing in the reverse order [78].

1.1 Problem Statement

Among different implementations of programming systems for Lucid [20] variants, GranularLucid (GLU) [37] introduced the concept of distributed demand-driven computation in the *intensional* programming environment. Unfortunately, the system was not flexible enough to accept further evolutions of Lucid, which has been and is still under constant change. Technically speaking, its demand-driven system was developed using a static and technology-dependent distributed environment by using the synchronous-only nature of Remote Procedure Call (RPC) [89]. Toward more dynamic and adaptive requirements, GIPSY uses a generic and technology-independent Demand Migration Framework (DMF) [38, 41], which offers a better infrastructure in terms of accepting both evolutions of the language and prospective goals of the framework. The DMF enables a generic infrastructure where various distributed technologies can be used in distributed migration of demands. The DMF includes a dynamic Demand Migration System (DMS) [38, 41] to propagate demands from one node to another across GIPSY execution nodes. Therefore, our demand-driven scenario is to have the generic DMF as our horizontal infrastructure where our DMS instances stands as its vertical components by applying DMF criteria.

This thesis extends the previous investigation of the DMS, which was implemented using JINI [62]. Previously, the DMS was implemented with the main focus on its connectivity in a heterogeneous distributed environment and demand propagation from one node to another. As the primary goal of DMF was a generic framework extensible enough to adopt arbitrary distributed middleware technologies and flexible enough to overcome the obstacle of co-existence of different versions, we developed another instance of the DMS (JMS-DMS) using Java Message Service (JMS) [90],

by combining different distributed paradigms which are available to this technology. Our primary goal was to develop a message-oriented and event-driven middleware to handle our demand-driven procedure. In addition to that, we studied and analyzed a multi-dimensional list of Quality of Services (QoS) to compare current instances of our DMF to benchmark the principles of high availability, scalability, maintainability, latency, flexibility, and etc.

1.2 Contributions

The main contribution of this thesis address the previously mentioned research problems through:

- analyzing the workflow of our current demand migration system from the Demand Generator point of view (see Section 4.5.1),
- investigating the current GIPSY internal implementation and determining the needs and requirements of the prospective multi-tier architecture,
- investigating and studying new requirements of our middleware such as having new types of demands (see Sections 4.4.1 and 4.5.2),
- implementing a message-oriented DMS by applying rationales and features of our generic DMF using Java Message Services (see Chapters 4 and 5),
- providing an elaborated list of quality of services for our comparative studies (see Section 3.3),
- implementing a simulator to provide an infrastructure for our comparative studies (see Section 6.1), and

- studying and comparing our implementation instances according to the list of quality of services (see Chapter 6).

1.3 Structure of the Thesis

This thesis is organized into seven chapters. Chapter 1 introduces the problem statement and contributions of this thesis. Thereafter, in Chapter 2, we introduce the background knowledge and related work we need to know in order to comprehend this research better. In Chapter 3 we present the methodology we are using in our design, implementation and testing stages. In Chapter 4, we study the general framework design of our system, and subsequently, the current and future detailed design of our architectures in different aspects. We described the implementations of both of our instances of the demand migration system in Chapter 5. In Chapter 6, we study their performance and system behavior in different aspects, and at the end we give conclusions on our research in Chapter 7.

Chapter 2

Background and Related Work

Let the beauty of what you love be what you do.

J.M.Rumi - Persian Poet (13th Century)

In this chapter, we present the background material that is necessary for the understanding of the core contributions of the thesis. At first, we briefly describe our experimental domain, which is GIPSY, and finalize our background section with information about two paradigms of Message-Oriented Middleware (MOM) and Event-Driven Architecture (EDA), which are used in our implementations of the Demand Migration Framework.

2.1 Background

2.1.1 Intensional Programing

Intensional programming, in the sense of the latest evolutions of Lucid [20], is a programming language paradigm based on the notion of declarative programming where the declarations are evaluated in an inherent multidimensional context space. Considering the context space being in most cases infinite, intensional programs are

evaluated using a lazy demand-driven model of execution called Eduction [47], where the program identifiers are evaluated in a restricted context space, in fact, a point in space, where each demand is generated, propagated, computed and stored as a identifier-context pair [21]. *Intensional* programming can be potentially used to solve widely diversified problems, which can be expressed using diversified languages of intensional nature. There also has been a wide array of flavors of Lucid languages developed over the years. Yet, few of these languages have made it to the implementation level [95, 57].

2.1.2 General Intensional Programming System

The Lucid language family has evolved since the beginning in terms of language syntax and semantics. However, some of the Lucid compilers and execution systems such as pLucid [47], GranularLucid (GLU) [37] could not cope with such evolutions. Therefore, the General Intensional Programming System (GIPSY) has been introduced to provide an infrastructure to accept such evolutions and contribute more on the required concept of distributed demand-driven execution. This has been done by the introduction of the generic Demand Migration Framework (DMF) in its execution environment.

The GIPSY project aims at the creation of a programming environment encompassing compiler generation for all flavors of Lucid, as well as a generic run-time system enabling the execution of programs written in all flavors of Lucid. Our goal is to implement a flexible platform for the investigation on programming languages of intensional nature, in order to prove the applicability of *intensional* programming to solve important problems.

2.2 Middleware

Puder et al. in [80] described general concept of middleware as: "Middleware offers general services that support distributed execution of applications. The term middleware suggests that it is software positioned between the operating system and the application. Viewed abstractly, middleware can be envisaged as a *tablecloth* that spreads itself over a heterogeneous network, concealing the complexity of the underlying technology from the application being run on it."

2.2.1 Message-Oriented Middleware

There are different types of middleware categorized by their method, needs of communication and interaction between distributed nodes, and among those, Message-Oriented Middleware (MOM) is one of the major forms of message-passing paradigms, which deal with intra-application communication across the network. MOM is currently one of the most sophisticated forms of distributed message passing.

According to [50], messaging is a technology that enables high-speed, asynchronous, program-to-program communication with reliable delivery. Programs communicate by sending packets of data called messages to each other. Message passing has been widely used in different applications. Message has been used both locally and distributedly in many small and enterprise projects. Some forms of message passing require exact physical address to connect to other distributed nodes, and some not.

MOM systems enable computerized applications that are physically separated or running on different hardware/software platforms to communicate with each other through interaction of messages [97]. This interaction can be done in a loosely coupled and asynchronous communication mode, as MOM supports working in both modes of synchronous and asynchronous communications.

As shown in Figure 1, the sender sends out its messages to a messaging-enabled

Figure 1: Simplified message-passing mechanisms in MOM

middleware. This figure shows a simplified MOM in a black-box form, as surrounded nodes and applications interact with each other regardless of the internal procedure inside the middleware. After receiving messages, the middleware starts notifying the receiver for new messages. Thereafter, it dispatches a message into a communication channel toward an appropriate receiver. The recipient receives that message, and after processing its request, returns the result to the middleware, and finally through that middleware, the original sender receives the response of its request.

G. Hohpe in [53] simply explains the asynchronous functionality of the MOM as similar to postal mail, where somebody writes a letter, sticks an address on the envelope, and puts the letter into a letterbox. The sender and recipient of the letter are decoupled, and the Post Office acts as a transporter. Each message is transported as a self-contained unit, and externally expresses only information related to the delivery. As an advantage of asynchronous communication in MOM, the recipient does not have to be present while messages are being transported. Later on in section 3.1.2, we will explain why in GIPSY we do not expect any discrimination for workers, as we do not stick any address on the envelope in a sense of identifying recipient of our demands before or at sending time. We only send out that we have such a request, and then, the DMS decides which Demand Worker is appropriate for this specific demand.

All operations of transition, migration, and dispatching in MOM follow certain requirements to fulfill the needs of a successful MOM system. We discuss them in

more detail in 2.2.1.1.

Java Message Services (JMS) [90] is one of the successful ways of implementing the MOM paradigm. We implemented the next version of our DMS by using JMS.

2.2.1.1 Benefits of MOM

MOM takes account of various benefits such as loose coupling, location independency, Quality of Services, and interoperability.

- Loose coupling

  Loose coupling (a.k.a. time independency) is an approach to the design of distributed applications that emphasizes the ability to adapt to changes [26]. Time independency is one of the most important characteristics of MOM. It is a design goal most sought by enterprise organizations that are implementing messaging systems. By loose coupling, every node communicates with each other in a much-decoupled way where their message migration takes place regardless of the location or state of a recipient node. The message sender and recipient do not have to be online at the same time, since MOM queues up messages when their recipients are not available. Both synchronous and asynchronous communication modes are available in MOM, and depending on the system requirements, either of those can be chosen.

- Location Independency

  Neither senders nor recipients know about the location of the other part of their communication. MOM uses queues or other means of communication to carry over message from one place to another, so actual locations of nodes are not important from the point view of the sender. Depending on the type of middleware, synchronous interaction requires blocking of the sender until it

receives the response from the recipient. However, in MOM these two nodes are decoupled, so they can continue working or waiting for the response at the same time.

- Quality Of Services

  In order to have a successful implementation of MOM, certain qualities of services such as availability, scalability, security, fault-tolerance, portability, etc. should be provided. We will discuss them in more detail in Chapter 7.

- Interoperability

  By its nature, MOM uses messages to communicate with other applications on distributed machines. Messages are not only bound to any network, operating system, or platform infrastructure, but also it is in a plain format that is understandable for any appropriate consumer or producer. Therefore, as long as messages are understandable by both sides (i.e., running appropriate execution node on the network), it is independent of any platform or operating system to interact with the middleware. This inherently portable character of MOM helps one to use the functionality of the execution nodes properly in any type of network.

2.2.1.2 Communication Modes and Domains

MOM uses either synchronous or asynchronous communication. This ability makes MOM dynamic and flexible in different conditions and scenarios. The fact that one of them should be used is directly bound to the requirement and specification of the system. We will explain detailed information about each of these two modes in 5.2.3 and 5.2.3.

According to [53], there are three types of messaging systems; message passing,

publish/subscribe, and message queuing. Message-passing like Remote Method Invocation (RMI) [51] is not applicable to our DMF as it supports only synchronous mode of communication, which according to our DMF rationales mentioned in section 3.1.2 is not desirable. As for this thesis, we provide an infrastructure where we can possibly have different types of communication domain, so we enabled both message queuing and publish/subscribe in the DMF (see 5.2.1.2).

2.2.2 Event-Driven Architecture

According to [48], Event-Driven programming has been independently developed in different applications. In order to demonstrate the inter-communication of modules and sub-modules, different data flow and structure diagram use event passing. This approach has been extended in client-server applications where both tiers interact with each other by passing and raising events. Recently, there are many distributed and local event-driven applications on both shelves of academia and the industry.

2.2.2.1 Rationale

According to [62], the Event-Driven architectural pattern may be applied by the design and implementation of applications and systems, which transmits events between software components and services. Like MOM systems, an event-driven system typically consists of event consumers and event producers. By occurrence of certain condition or change of the state of an object or element, a specific pre-defined event happens. As soon as the EDA-enabled middleware receives an event, it notifies a consumer for the presence of such an entity (i.e., event) in its system. Event consumers subscribe to an intermediary event manager, and event producers publish to this manager. When the event manager receives an event from a producer, the manager forwards the event to the consumer. If the consumer is unavailable, the manager

can store the event and try to re-forward it later. We can see both of our instances follow EDA paradigms for their demand-migration procedure.

2.2.2.2 Terminology

Here we introduce the list of terminology used in EDA in order to comprehend this architecture better.

Event In [97], events are referred to as "an occurrence in one application or component that others may be interested in knowing about." An event can be:

- Sending a request or demand (manually or automatically)
- Change of the state of an object
- An occurrence of certain condition
- Throwing an exception or error in the system

In our GIPSY DMF, demands can be easily considered as events (see 3.1.4). More detailed specifications about the types of events are discussed later in Chapter 4.

Producers Producers are those who publish events (i.e., Demand Generator in our DMS) across the network. Depending on their communication and architecture, producers may or may not address an event to any specific receiver.

Consumers Consumers are those who receive the events from the system (i.e., Demand Worker in our DMS) upon occurrence of an event. After receiving events, they do some appropriate action to process the demand. They are not necessarily aware from which producers they receive events.

Event Channel Subscription Upon their interest, consumers subscribe to interesting subjects (i.e., Event Channels), so whenever a new event appears into one of those communication channels (synchronous or asynchronous) they receive it. In chapter 5, we explain each event channel in our DMS instances.

2.3 Related Work

We divided our related work in two sections: (1) those who are related to our technology implementations, and (2) those who are related to our benchmarking approaches.

2.3.1 Implementations

Related to our implementation approaches, we investigated the following projects:

TERA [82] is an Event-Driven Architecture [92] system designed to offer an event dissemination service for very large-scale peer-to-peer systems. Exactly like DMS, TERA enables topic-based message selection, but only works with the publish/subscribe communication mode. Compared to their system, in addition to the Publish/Subscribe mode, we provided Point-to-Point communication, which later on we explain why it is more interesting for us. They both use certain kind of identifiers in their message selection and subscription management. In [18], very abstract information about having decoupled middleware by using JMS and JavaSpace [49] was mentioned, and at the end asserted that JMS is designed for information delivery, whereas JavaSpace can be called an information-sharing infrastructure. In our research, we tried to investigate the limitations of each of those, and point out their strengths in different situations.

In [39], similar to our implementation, NASA have used JMS for their asynchronous messaging among their multi-tier portal architecture. Implementation-wise,

we both have used facade design pattern in order to implement a lightweight interface for our entity transitions. They use JMS to provide services and interact between their intra-modules (e.g., transfer a new image downloaded from Mars to another module of their portal), and we used JMS for the purpose of interaction between our GIPSY modules. These kinds of research show that MOM and particularly JMS have been used vastly in different mission-critical as well as enterprise projects.

MOM-G [59] is a MOM based on the Grid environment that transits data in both synchronous and asynchronous mode using XML [54] with SOAP [94] convention. Even though the infrastructures of our cases are different, but in a very high-level observation, these two implementations look similar as the packets are transferring from one node to another. However, in our case, we use JMS Object messages, which can handle any kind of `Serializable` Objects, but in their case it is only XML. This limits the plugability of a system based on this middleware, as we cannot attach any code or object to the demands, whereas in our case it is necessary.

2.3.2 Benchmarking Approaches

Considering the related benchmarking experiments in distributed environments, we studied the following projects:

In [30], an empirical methodology to evaluate the quality of services of distributed system was provided. We have taken a similar approach in our investigations to provide information about the maximum sustainable throughput [30], latency, and elapsed time for batch messaging of our systems. We have extended their approaches by evaluating more qualities of services and challenging our failure points to study our scalability.

Bo Lu in [21] analyzed three approaches adapted in GEE implementation by addressing two important issues of performance and resource management in both

standalone and clustered machines. However, his focus of analysis is on the performance of the GEE implementation, whereas in our case we studied the DMS which is basically one step further in the process of demands migration as DMS uses generated demands from GEE.

There are many academic projects working on the evaluation of performance and throughput of two different distributed systems. Throughput of different JMS providers have been evaluated in [22] in different aspects. In this research, we are also evaluating the throughput of our DMS instances in facing different loads and types of demands. In the benchmarking of closer types of distributed systems, according to [84], S. Rooney et al. provide a feasible study of load balancing and performance of scalable asynchronous and tuple-oriented messaging. Both DMS instances use tuple-oriented demands to carry different elements to identify uniquely every single demand. In [69], performance of four commercial JMS providers is compared in terms of their scalability and persistency. In this thesis, we provide stress/batch messaging in order to test our system scalability as well.

In [68], the author simulated the delivery of messages using the epidemic protocol in ad-hoc and wireless networks. Similar to our benchmarking approaches, they have tested the delay time and buffer size for both persistent and non-persistent JMS messaging, but in our case, in addition to those tests, we have included more experiments such as persistency, priority, portability, and etc. They mainly worked on the ad-hoc system, whereas in our cases we did it in a traditional distributed system.

Chapter 3

Methodology

Even after all this time the Sun never says to the Earth,
'You owe me.' Look What happens with a love like that,
It lights the whole Sky.
Sh. Hafez - Persian Mystic and Poet (14th century).

In this chapter, we discuss our methodologies in different aspects of design, implementations, and finally our research investigations in terms of different quality of services. First, we briefly rationalize the framework criteria and explain the core design of our system. Later on, we highlight the needs of having different instances of our migration system in terms of implementation, and finally, we complement our investigations with comprehensive experimental studies on our available DMF instances.

3.1 Design

We discuss our rationale and design approaches in the framework and subsequently in the DMS. In order to explain the scenario in more detail, we explain what we mean by demands in GIPSY, and how we deal with them as messages and events.

We also present our design from the highest level of abstraction (i.e., DMF) to the lowest level of internal procedures of DMF submodules. This can be found in more detail in Chapter 4.

3.1.1 Demand Migration Framework

3.1.2 DMF rationale

In order to provide a realistic technology-independent Demand Migration System, in [38, 41], we initially introduced an elaborated list of important rationales that we applied to all of our implementations to make them as much generic and similar as possible. In addition to the achieved simplicity, this similarity eventually increases the level of integration among GIPSY modules. All of the following rationales are implicitly addressed in Chapter 5.

- Platform interoperability

  GIPSY programs are potentially evaluated on multiple platforms [78]. Hence, the DMS serving the GIPSY nodes (executing on separate machines) should be able to deal with the machine boundaries and the diversity of the platforms executing the GIPSY nodes, i.e., the DMS should be able to connect the GIPSY nodes executing on various operating system platforms using different middleware technologies available on these different platforms.

- Once and only once delivery semantics

  The GIPSY nodes run independently from each other [78]. Hence, the DMS needs to be able to connect these artifacts at any time and to assure once and only once delivery semantic, i.e., no demand or result could be delivered to a wrong node or duplicated.

- Asynchronous Communication

  Since the GIPSY nodes are not necessarily synchronized — they run independently and their lifetime is not synchronized, the DMS must also 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 [78], the DMS should not discriminate them in terms of importance. In DMF level, while dealing with demands entities, we do not consider any discrimination, but in the implementation level, many efficiency-related considerations such as prioritizing certain types of critical demands are to be tackled by other parts of the GIPSY.

- No Workers Discrimination

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

- Secure Communication

  Since the GIPSY nodes are located on different machines, we need to use security mechanisms [41] to authenticate the identity of the DMS objects. The DMS should integrate a secure mechanism.

- Fault-tolerant Demand Migration

When objects are distributed across process boundaries, the objects can fail independently [41]. Similarly, in a distributed system the network can be interrupted or the system can be partitioned into disconnected parts, and components (nodes) can run independently, even if others have failed [41]. 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 permanently 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. In the case of DMS failure, it should be able to re-start and use the permanent demands storage mechanism to continue without losing 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 a platform independency, i.e., the DMS must be flexible and structurally generic to work with most of these distributed computation technologies and implementation platforms.

- Hotplugging

  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 DMF should not be bound to any distributed execution technology and be generic enough to allow the use of other technologies.

3.1.3 Demand In GIPSY

In GIPSY, a demand can be considered as an event. Referring to the definition of demands in *intensional* programming, a demand is a request for the evaluation of a certain identifier in a given multi-dimensional context . It can be either *intensional* or *procedural* (functional), which we discuss more in detail in section 4.4.1. Sometimes this kind of evaluation is directly bound to the functionality of another method, which might be written in C++, Java, or potentially any other programming language.

In order to explain the situation in a very simplified way, in the given example below, the value of the identifier $\mathcal{A}$ is requested. As shown here, a three-dimensional context $[\mathcal{X} : 0, \mathcal{Y} : 3, \mathcal{Z} : 2]$ is defined over the three dimensions $\mathcal{X}$,$\mathcal{Y}$, and $\mathcal{Z}$, placing the computation of $\mathcal{A}$ at coordinates [0,3,2] in the multidimensional context.

$$\textit{Dimensions}= \mathcal{X}, \mathcal{Y}, \mathcal{Z}$$

$$\textit{Coordinates=0,3,2}$$

By looking up the values of $\mathcal{A}$ in these three dimensions in the given coordinates, we can find out the actual value of identifier $\mathcal{A}$ in that specific context. For example,

$[\mathcal{Z} : 2]$ associates the context in the dimension $\mathcal{Z}$ of this identifier, where its coordinate is 2. The same rule accords to the rest of dimensions (i.e., $\mathcal{X}$ and $\mathcal{Y}$).

In this thesis, we deal with demands in a very abstract way. We do not dig into the semantics of the multi-dimensional context, as there were done extensively in other research in the GIPSY project [64, 63]. By its abstract nature, the DMF and its implementations simply migrate demands between nodes and do not need to neither verify their correctness nor apply any sort of computation on the demands during their migration procedure. Correctness verification and computing are done at the level of the run-time system by the Demand Generators (DG), which is outside of the scope of this thesis.

$$\mathcal{A} = \mathcal{B} + \mathcal{C};$$
$$\mathcal{B} = 2;$$
$$\mathcal{C} = f_2(...) + \mathcal{D};$$
$$\mathcal{D} = 4;$$

To elaborate furthermore, the DG looks up into the definition of the identifier $\mathcal{A}$ at the runtime program definition dictionary (called GEER, for General Eduction Engine Resources) and finds out that it needs to initiate two more demands (see the next page) for the computation of $\mathcal{B}$ and $\mathcal{C}$. For example in the given example above, $\mathcal{A}$ is a Lucid identifier whose results is bound to the result of computing both $\mathcal{B}$ and $\mathcal{C}$. Given the situation where we have all required resources, one can have the result of $\mathcal{B}$ by simply looking up into the value of $\mathcal{B}$ in its multi-dimensional context space. As $\mathcal{B}$ is declared as a constant ($\mathcal{B} = 2$), its value does not vary in any dimension. Therefore, no further computation is required to compute $\mathcal{B}$. However, for the other identifier (i.e., $\mathcal{C}$), the result is bound to the result of the computation of its procedure $f_2(..)$ (e.g., $f_2(..)$ as an example of a procedure call), thus raising the need for a procedural demand that can be computed by a Demand Worker (DW).

GIPSY allows such procedures to be defined potentially in any procedural language, and the DWs are assumed to have access to a compiled version of such procedures, as embedded in the corresponding GEER [70, 85]. The demands (intensional or procedural) are generated by the DG , and consumed by either another DG or a DW, with the DMS acting as middleware for their transparent migration and taking no part in verifying or computing them.

3.1.4 Demand vs. Event

In order to provide a flexible infrastructure for distributed demand-driven computing in GIPSY we are investigating on the DMF and providing various instances of it using different middleware technologies. We currently have two instances (one based on JINI, another based on JMS), and we are still investigating on different architectures and paradigms to implement other instances of the DMF. We review the benefits of using each or combination of them to provide a middleware which can be a good environment to correctly implement our DMF requirements.

As part of the terminologies of EDA in 2.2.2, we considered different terms such as event, notification, producer, consumer, subscriber, filters, and channels in the implementations of our DMS. In GIPSY, DGs are considered as producers of demands (i.e., f2(..) in the preceding example), which initiate the procedural demands, which can be executed either remotely or locally by a DW. The DG sends out the demand to the DMS, and it routes the demand to an appropriate channel which is subjected respectively according to the subject or/and content of the demand. Each event (i.e., demand) can be differentiated based on their subject or content. The DMS transfers demands across the GIPSY nodes from one point to another through means of communication which are either queues or topics.

As soon as a demand arrives to the DMS, DMS invites DGs and DWs to consume

it. These demands request an intensional evaluation by another DG (in the case of intensional demands) or procedural computation by a DW (in the case of procedural demands). Therefore, by implementing such a concept, the GIPSY-DMS can be considered as using an Event-Driven Architecture.

It is important to explain that the resulting value of each GIPSY demand is referentially transparent in the sense that the response of the evaluation of a specific identifier in a specific context always yields the same value. Therefore, no matter when it is initiated, it should always return the same result. In order to improve the performance and decrease unnecessary computation, the DMS permanently stores the result of each demand in what is called the *demand store*. In case that the same demand is generated twice, the value is retrieved from the *demand store* instead of computing a new one.

In our future work, we would like to investigate the impact of our storing mechanism in Demand Store to see how this I/O mechanism changes the performance of our system in low and high loads of work.

Depending on the communication mode, distributed notification can be used to notify DGs and DWs. DGs and DWs (consumers/subscribers) subscribe themselves to an appropriate channel, which can be either a specific topic or queue. In some practices flooding [71] has been used to send and forward all the notification to all the subscribers. Such behavior is not desirable in GIPSY. Flooding causes many unnecessary transactions to generate and migrate demands to all the consumers, whereas in GIPSY architecture, a demand should be executed once and only once. In GIPSY-DMS, filtering is also enabled to permit the consumers to select specific kinds of demands. For example, DWs shall only receive procedural demands, and thus shall be able to filter out intensional demands, which they cannot compute. As one of the requirements of EDA, we enabled demand selection based on their contents, size, or

subjects, to improve load balancing, availability, and scalability in demand migration.

3.1.5 Demand vs. Message

Message passing has been used widely in different application domains. In [48], there are many examples of using message passing in diagrams and applications. In such a system, messages transfer from one application to another, and upon reception of the message, an action is triggered. It has been used in different design patterns to facilities push and pull (e.g., Observer Design Pattern) [66] from one class to another. According to [48], the purpose of a messaging system is to get events (messages) from event generators (senders) to handlers (receivers) in situations where the senders and receivers are in different physical locations or running on different platforms.

Formerly in previous *intensional* programming systems, messaging was only synchronous and highly coupled in a sense that the producer (i.e., DG) and consumer (i.e., DW) were blocked until the computation ended, and the producers send their demands to a specific consumer in the network. However, in GIPSY, there is no discrimination in choosing DW from the scope of view of the DG. In [48], Message-Oriented Middleware (MOM) is referred to as the most sophisticated enterprise messaging system for decoupled components.

By pointing out some problems of having static and highly coupled inter-messaging in GLU [73] and other *intensional* programming systems, GIPSY uses a loosely coupled middleware by using asynchronous message-passing paradigm such as MOM.

After studying different commercial and academic implementation of MOM, Java Message Services (JMS) [52], from Sun Microsystems, was chosen. JMS is a set of interfaces and associated semantics that provides a common way for Java programs to create, send, receive, and read an enterprise messaging system's messages. Message portability is thus strong in JMS, which suits well with the fact that the migration

of messages is abstract from the respective scope of view of both DG and DW. These GIPSY nodes only use the JMS interface to connect and close a connection and sessions without knowing anything about the internal implementation of the middleware.

After starting the connection, the DG writes its demand on one of the designated JMS destinations such as queue and topic. The interaction among each of these modules can be done in both synchronous and asynchronous mode.

As explained above, as we have to use many of MOM features such as asynchronous/synchronous messaging across our communication nodes in a totally loosely-coupled way, we consider DMS as a Message-Oriented Middleware.

3.2 Implementation

In the course of this research, we applied the DMF generic architecture for migrating objects to our design of a DMS based on using JMS, and built a JMS version of the DMS to transport demands in a heterogeneous and distributed environment. As we will discuss in Chapter 5, both implementations of our DMS (i.e., JMS and JINI) follow the layered conceptual architecture implied by the DMF. Detailed information about both implementations is given in Chapter 5.

3.3 Experimental Investigations

One of the major goals of this thesis was to investigate different scenarios and conditions of demand migration in GIPSY environment. These investigations required the implementation of different versions of DMS and their comparison them in different circumstances. This requires a comprehensive list of criteria from the point of view of GIPSY to make the comparison of these instances significant.

Performance is one of the most important key elements in choosing a distributed

system's middleware. In both worlds of industry and academia, there are many research and benchmarking investigations to show off middleware capabilities comparing to other competitor solutions.

Comparing two systems requires an excellent in-depth knowledge of their detailed designs, implementations, and even their specification and requirements. Performance is not an independent factor to be compared solely regardless of different critical elements of their system requirements. There are many important criteria building up dimensions of qualities of service (QoS), which can demonstrate adequate performance of a system. Thus, QoS in this thesis explains how DMS instances are required to work, and how well they are performing.

Every single solution may be suitable for a specific scenario, so it is wise to have domain-related solutions rather than a general solution for all possible purposes. Therefore, in our GIPSY lab, in the domain of *intensional* programming, we developed a generic Demand Migration Framework (DMF) to facilitate required infrastructure for the purpose of distributed demand-driven communication.

Each of these two implemented systems (i.e., JINI-DMS and JMS-DMS) has its own characteristics. Therefore, comparing them requires deep knowledge of their design, implementations, specification, and requirements. We set some major criteria and goals for our generic DMF, and test its two instances in different scenarios against these criteria. In the following, our major criteria in terms of QoS appear, and later on in Chapter 6, we will mention how we investigate them.

Here, we list many important elements for the evaluation of the QoS in distributed systems, but among them, we chose those which we could test regarding their level of importance in the GIPSY environment and our limitation of time in the course of this research.

Our two DMS instances respectively use the JINI and JMS technologies as a

multi-platform transport protocol, i.e., it is able to connect machines with different operating systems. This transport protocol is implemented in the form of Transport Agents (TA). Each one of these 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. By having different implementation, we can benchmark many scenarios of demand migration in different execution environments. In our experimental investigations, we evaluate our DMS instances in terms of their performance, overhead, implementation, integrability, throughput, plugability, as well as limitations.

3.3.1 Simulator

GIPSY is a complex system, where different subsystems are designed and implemented as loosely-coupled components. In the course of this research, we have encountered the need to test our currently implemented components, while other where under development. Therefore, we designed and implemented a light version of GIPSY, a prototype called GIPSY Simulator to help us gather information about the functionality of the entire GIPSY system. We discuss about its features and functionalities in more detail in Chapter 6.

3.3.2 Quality Of Service

A network that supports QoS is a network that presents its capabilities to the user and allows them to make choices for the service they receive [34]. These qualities are more precisely definable if the infrastructure is static in the sense that it does not change dramatically over the time, but in real-life, network and distributed systems are inherently dynamic. For example, if a system provides 99.99% availability for up to 16 nodes, it may not behave the same QoS if the number of nodes exceeds

1000. Defining these limitations is very complicated and challenging, as we generally like to expect a distributed system that supports unlimited number of nodes. As our infrastructure at the university is limited, we can only experiment our actual system behavior in a smaller scale whereas in some scenarios we required thousands of nodes. For the time being, we narrowed down our investigations to more limited number of nodes to see the general system behavior of the middleware. We made our investigations on the current limited infrastructure, and found out the guaranteed and possible quality of services of our system in this limited context.

Despite these limitations, our experimental investigations help us analyze our two DMS instances in more detail, and provides us additional information about the system behavior of each version in different scenarios. Therefore, we can expect to use each of them in the condition they perform better, permitting better flexibility of use for computation using GIPSY. Coexistence of these two framework instances in every environment is costly, in the sense that it requires more time to maintain, modify, comprehend, and extend. Even with considering such a cost, performance of the entire system is by far better than the situation that only one single system is available. For example, we can find out that one instance is suitable for demand migration of demands larger than a certain size, but the other one provides better performance and throughput for smaller ones. We can use either of them in different scenarios by enforcing the use of a designated one for certain demands specifications. By identifying these QoS, we can investigate more on parts where these systems behave poorly. We should identify those points and investigate on their improvements as parts of our ongoing and future investigations.

In order to compare the performance of these two versions, we cannot just rely on a result from a test case or two. We cannot just run these systems in a scenario where they generate and compute hundreds of demands remotely, and label the one

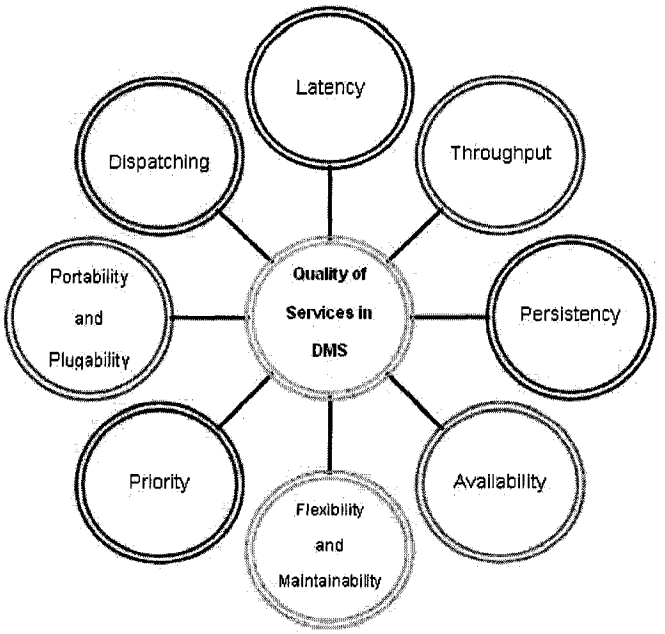

Figure 2: Multi-dimensional list of Quality of Services in DMS

which provided faster result as a better solution. We need to find out important elements, which are mandatory and critical in our framework, and eventually build up our test cases on top of that. The out-coming results from such comprehensive tests provide accurate and mature conclusion about the future of each versions of DMS. Considering only faster response time as better performance does not provide us a better solution for most cases. As in our environment, response time is important but that is not the only criterion. Other important factors of QoS are important to benchmark our system, so we should deal with QoS as a multi-dimensional aspect to provide as much experimental information as necessary. By having multi-dimensional QoS, we can find out which DMS version is performing better in specific dimensions.

By reviewing the related literature, generally QoS for middleware consists of qualities such as latency, throughput, availability, concurrency, persistency, maintainability, filtering, flexibility, indexing, durability, portability, and priority. However, in this thesis, we just experiment with some of them which are depicted in Figure 2.

At the end, in the experimental investigation chapter, by observing the results from those tests, we can conclude which version in suitable for certain conditions. Here we introduce each of these dimensions, and later on in Chapter 6, we discuss them in more details with comprehensive information and test results.

Availability

One of the most important criteria in our DMS is the availability. During our demand-migration procedure, a node which hosts the DMS may suddenly go down or up due to some unexpected exceptions, failures, or errors. Regardless of the network availability factor, we would like to compare these two instances against this factor, and see which one of them starts and establishes its services faster.

Latency

The time elapsed from the point a node sends out a message to the point another node receives that request is called latency [30]. Latency is often used to mean any delay or waiting that increases real or perceived response time beyond the response time desired.

In our system, this applies to the time a DG sends out a demand to the point a DW receives it. Latency might vary according to the network configuration and load, number of DW and DG, filters, and etc.

The most difficult issue in measuring the round-trip is time. As both DW and DG may be distributed across the network, having synchronized time is a difficulty. One of the approaches to latency measurement is setting the timestamps in each message, and acknowledging it back to the DG for the further analysis. In 6.3.3, we explain in detail how we are going to test this factor in our system.

Throughput

Average number of messages that can be delivered in a period is considered as throughput of a system. It is the most common performance metric used to evaluate the capacity of a messaging systems [30]. Some previously published tests in [83] have measured throughput by measuring how quickly messages can be sent, without ensuring that messages are received at the same rate. A side effect of this technique is that messages may be produced far faster than they can be consumed. In 6.3.5, we will explain the detail information about our test cases for this specific element.

Persistency

Persistency is an important issue, as the DMS might go down at any time. In order not to lose any demand, we have to check how persistent our system is. In case of any sort of failure or breakdown of the DMS, there might be demands waiting to be served. We do not want to lose those waiting demands in our buffers or queues. The DMS needs to provide persistent storage of demands. In addition, the DMS also stores the values resulting from the computation of all migrated demands. By providing a permanent-storage mechanism for demands, DMS, allows for the results to be permanently stored, thus enabling computations to be stopped and resumed later without having to recompute previously computed results. We explain our test cases for persistency in 6.3.6

Flexibility and Maintainability

As said before, the DMS is a part of an ongoing project, which should be a dynamic and flexible infrastructure for any kind of future maintenance and extension. Because GIPSY itself is under investigation, we should expect to face future features to be

required in the system. Therefore, both instances of DMS should provide a properly-developed structure to be maintained and expanded easily. Adding, modifying, or even removing some features should be considered as part of its dynamic and active architecture. We divide flexibility into two parts: configurability, and ease of use. We evaluate our two DMS instances from the viewpoint of these qualities. Detailed information about this aspect of our benchmarking comes in 6.3.7.

Priority

Priority is a very important feature in DMS, as different demands should have different level of priorities. As we would like to work with different types of demands (see 4.4.1), we should enable the feature of having different levels of priority in order to expedite a demand, which has a higher level of importance.

Portability and Plugability

As mentioned in 3.1.2, portability and plugability are important criteria in DMF rationales. Therefore, we include them in our major elements of our QoS. Plugability is important in the sense that our middleware as an external application can be used in a plug-and-play mode in different operating systems or networks.

3.4 Summary

In this chapter, we discussed our methodology and approaches to design, implement, and test our middleware. We began with the presentation of our criteria and rationale and then we discussed how we can deal with the procedure of demand migration in our design and implementation. At the end, we provided a list of QoS that we would

like to investigate our systems against. In the next chapter, we will present more details about the design aspect of our Demand Migration Systems.

Chapter 4

Design

Out beyond ideas of wrongdoing and rightdoing,

There is a field. I will meet you there.

J.M.Rumi - Persian Poet (13th Century)

This research stands inside an existing research and development framework. Consequently, our design and implementation is constrained into the existing design and code base. This chapter aims at describing the existing GIPSY Demand Migration Framework and explain how our solutions have been integrated in such a framework.

4.1 Demand Migration Framework

The DMF is a generic scheme for migrating objects (GIPSY demands) in a heterogeneous and distributed environment specified by the GIPSY nodes and by the GIPSY tiers nested in these GIPSY nodes. Thus, the DMF establishes a context for performing demand-migration activities, where the migrated entities encapsulate embedded functions and data pertaining to the processing of these demands.

4.1.1 DMF Architectural Model

The DMF exposes a layered-structured architecture [38], which helps the functionality of the system to be implemented in several layers, and abstracting the clients of the demand migration from its implementation intricacies. Figure 3 represents an abstract conceptual view of the DMF. As it is shown by the layered structure, the largest circle depicts the entire GIPSY, and the double-lined inner circle depicts DMF. GIPSY is represented as a set of operational nodes — workers (DWs) and generators (DGs), those being the communication end points, and the DMF acting as a communication intermediate between them. The DMF consists of two main functional layers called `Demand Dispatcher` (DD) and `Migration Layer` (ML). The Demand Dispatcher (depicted by a bold-lined circle in Figure 3) is an object-based storage mechanism able to dispatch objects to their recipients. The `migration layer` (depicted as a dark grayed layer on top of the Demand Dispatcher) is the layer performing object migration from the Demand Dispatcher to the recipient `Demand Worker` or `Demand Generator`. The `migration layer` makes the communication in the heterogeneous distributed environment possible. The DMF relies on these two functional layers to form an asynchronous communication system similar to the persistent asynchronous message-passing systems [67], i.e., the messages are permanently stored and delivered upon request.

Moreover, the demand dispatcher layer establishes the context of a centralized demand propagator that consists of two layers — `Demand Space` (DS) and `Presentation Layer` (PL) (see Figure 3). The `Demand Space` layer defines a context of internal object-based storage mechanism. The `Presentation Layer` is an abstract layer on top of `Demand Space` that makes the `Demand Space` functionality transparent and generic. The DMF establishes a complex model for communication. Therefore, dividing the functionalities into several sets, where the inner functions are tightly coupled,

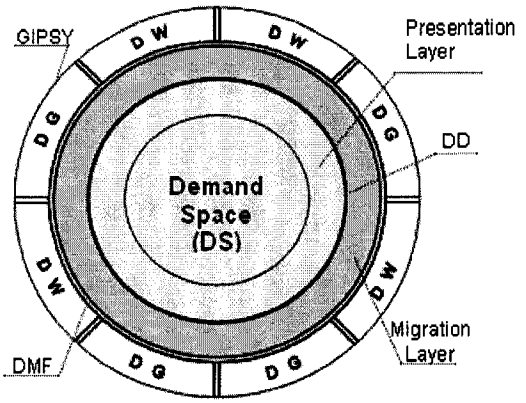

Figure 3: GIPSY Demand Migration Framework (DMF)

but highly independent from the other layers, is very appropriate. This technique helps to address the problem sets separately and to reach for higher upgradeability and flexibility through modularity.

4.2 Demand Migration System

The DMF is a generic framework solution to the problem of migrating demands in a heterogeneous and distributed environment. Hence, the DMF does not impose any technologies or platforms, but rather provides guidelines to design a DMS (Demand Migration Systems). Our first two DMS applications (i.e., DMS instances) are based on the distributed technologies of JINI and JMS.

4.2.1 DMS Layers

Figure 4 represents the layered structure of our first DMS generic architecture derived from the DMF. In this Figure, the `Demand Dispatcher` (DD) and the `Transport Agents` (TA) (see Section 4.2.3) are subsystems of the DMS, inherited from the DMF framework. The Demand Dispatcher consists of two contributors — `Demand`

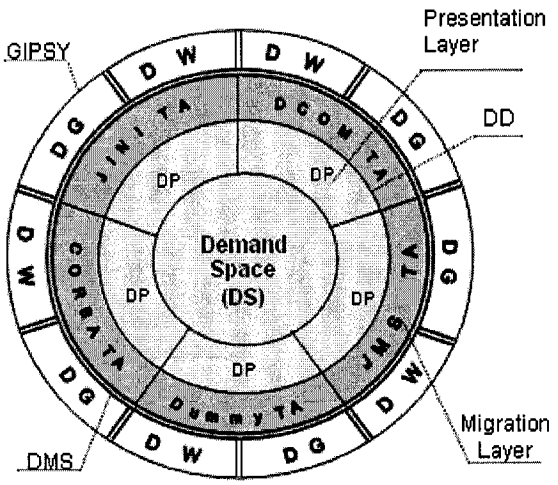

Figure 4: GIPSY Demand Migration System (DMS)

`Space` (DS) and `Dispatcher Proxy` (DP). Whereas the `Demand Space` comes directly from the DMF, the Demand Proxy is a design solution instance to the generic `Presentation Layer` (see Figure 3). The `Presentation Layer` consists of multiple demand proxies, each being associated with a `Transport Agent`, i.e., a `Demand Proxy` is a `Demand Dispatcher`'s entry point. The Demand Dispatcher has multiple Demand Proxies and one single `Demand Space`. The `Migration Layer` (ML) is presented as a set of `Transport Agents`, each one based on a different distributed technology (e.g., JMS, JINI). The Dummy `Transport Agent` simply exposes the `Demand Proxy` to the local and Demand Generator instances, bypassing unnecessary remote communication procedures. In Figure 4, DWs and DGs are grouped into pairs. Each DG-DW pair relies on a `Transport Agent`, which could be common to both. This is just one of the many possible run-time arrangements.

4.2.2 Demand Dispatcher Layer

In general, the `Demand Dispatcher` layer maintains a pool of demands to be processed. The following elements describe the `Demand Dispatcher`'s internal structure.

DP (Dispatcher Proxy) The `Dispatcher Proxy` inherits the `Presentation Layer` (PL) from the DMF (see Section 4.1.1), i.e., it works as a proxy for the `Demand Dispatcher`. The `Demand Dispatcher` relies on it to expose functionality to its clients. The clients are `Transport Agents`, `Demand Generators` and `Demand Workers`. All the `Demand Dispatcher`'s clients are assigned with a unique `Demand Proxy`. The `Demand Generators`, `Demand Workers` and `Transport Agents` use the `Demand Proxy` functions as their own, in their local address space, thus hiding the complexity of a possible remote collaboration with the `Demand Space`.

DS (Demand Space) The `Demand Dispatcher` 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 `Demand Space` 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 [28].

4.2.3 Migration Layer

The `Migration Layer` establishes a context for migrating objects among the GIPSY tiers and nodes. The `Migration Layer` provides a transparent form of migration. The `Migration Layer` refers to communication between computers and its architecture is based on the Open Systems Interconnection (OSI) Reference Model [38, 36]. In addition, the `Migration Layer` provides an architectural structure, forming a multi-platform transport protocol that is able to connect machines with different operating systems. The `Migration Layer` focuses on the use of `Transport Agents`, which are special kind of autonomous messengers (see Figure 4).

TA (Transport Agent) `Transport Agents` are based on distributed technologies whose architecture influences their implementation. `Transport Agents` differ in their structure and implementation, but they all expose the same interface to the Demand Generators, Demand Workers and Demand Dispatcher. Thus, despite the distributed technology diversity, their services are transparent and homogeneous with regard to their API.

TA Interface When a `Transport Agent` starts, it plugs into the system by connecting with the Demand Dispatcher and exposes its interface to Demand Generator and Demand Worker instances. Actually, the DWs and DGs listen constantly to DST for newly plugged `Transport Agents` and when the latter appear, they connect to them. Thus, DWs and DGs must adhere to the `Transport Agent interface` in order to connect to that `Transport Agent`. Figure 4 depicts DGs and DWs on top of compatible `Transport Agents`.

4.3 GIPSY Multi-Tier Architecture

At the GIPSY's core is a multi-tiered architecture, where the execution of the GIPSY programs is divided into four different tasks assigned to separate tiers. The GIPSY tiers spawn separate processes that communicate with other processes from the same or different tiers by using demands. A GIPSY tier is an abstract means of a computational unit that exposes a set of concrete GIPSY components that collaborate with each other to achieve program execution.

Note that we do not intend to describe the complete functionality of each tier in this thesis as it is out of its context, and we have not fully implemented these during the course of this research. As we have studied our current infrastructure, we have investigated on the multi-tier architecture as it provides more control on all our

totally decoupled GIPSY modules.

Demand Store Tier (DST) Instances of this tier represent a middleware that exposes certain functionalitiesto the other tiers of the DMS. The latter is a communication system that connects all GIPSY tiers via demand migration [14, 38]. In addition, the DMS provides a persistent storage of demands and their resulting values, thus achieving better processing performance by not having to re-compute the value of those already computed and stored demands. As initially mentioned in 4.2.1, The DMS architectural model consists of two major components — `Demand Dispatcher` (DD) and `Transport Agents` (TAs). They both run independently, but form together the overall behavior of the DMS, where the Demand Dispatcher acts like an event-driven message storage mechanism that uses the `Transport Agents` to deliver the demands from their origin to their destination, and their corresponding results in the reverse direction. The DMS relies on these two contributors (DD and TA) to form a message-persistent communication system [15].

Demand Generator Tier (DGT) Instances of this tier encapsulate a `Demand Generator` (DG), which generates demands using an `Intensional Demand Processor` [14, 41] that implements the eductive model of computation. The generated demands can be further processed by other DGT instances or by the DWT instances (see below). The demands are migrated between the tiers via the DST [15].

Demand Worker Tier (DWT) Instances of this tier encapsulate a Demand Worker (DW). The latter can process *procedural demands*, i.e., demands for a procedural function call from the intensional program [14, 41]. A DWT instance encapsulates a processor and a pool of procedural demands. The latter represents a compiled version of all the procedural demands that the `Demand Worker` is able to respond

to [15].

GIPSY Manager Tier (GMT) Instances of this tier enable the registration of GIPSY nodes to a GIPSY instance (a set of interconnected GIPSY tier instances), and the allocation of various GIPSY tiers to these nodes. A GIPSY node is a computer that has registered in the GIPSY network as a host of one or more GIPSY tier instances [15].

4.4 GIPSY Demands

In GIPSY, demands represent the means of communication between tier instances. Moreover, a demand is considered to be a request for the value of a program identifier (a Lucid identifier or a procedure identifier embedded in a hybrid Lucid program) in a specific context of evaluation. GIPSY specifies four demand types - *intensional*, *procedural*, *resource*, and *system* demands. During their life cycles, demands can transit over three different states - *pending*, *in process*, and *computed*. Both demand types and demand states, together with a unique demand ID, are used by the DMS for their identification, unification, and traceability.

4.4.1 Demand Types

Initially there were two types of demands in the GIPSY: *intensional* and *procedural* demands. However, in order to strengthen the system behavior of the DMS in the GIPSY multi-tier architecture, in the course of this research, we have introduced two more types of demands to the system. These new ones are `resource` and `system` demands, which are meaningful in different situation and scenarios where additional resources are required.

Intensional Demand A demand for the evaluation of a Lucid program identifier, given a certain context. *Intensional* demands are created and further processed by the `Intensional Demand Processor`, the main component of the `Demand Generator Tier`. Intensional demands have the form:

{`GEERid`, `programId`, `context`}

where `GEERid` is a unique identifier for the `GEER` (i.e., the compiled program) that this demand was generated for; `programId` is an identifier declared in this `GEER` (in this case a Lucid identifier); and context is for the context of evaluation of this demand.

Procedural Demand A demand for the evaluation of a certain procedure originally written in a procedural language, as part of a hybrid GIPSY program or a method in an Object-Oriented Intensional Programming Language. *Procedural* Demands are generated by the `Demand Processor`, and processed by the `Procedural Demand Processor` of the `Demand Worker Tier`. Procedural demands are generated by the `Demand Processor` of the `Demand Generator Tier` as it encounters procedural function call nodes while traversing the Abstract Syntax Tree (AST) to process demands. The procedural demands, after creation, are accumulated in the `Local Demand Store`, which will use its `Transport Agent` to have the procedural demands (as well as the intensional demands) migrated to a `Demand Store Tier`. A Demand Worker Tier will then request procedural demands from the `Demand Store Tier`, which will be processed and the result then migrated back to the `Demand Store Tier`, and then to the `Demand Generator Tier` that generated it. A procedural demand has the form:

{`GEERid`, `programId`, `Object params[]`, `context`, [`code`]}

where `GEERid` is a unique identifier for the `GEER` (i.e., the compiled program) that this demand was generated for; `programId` is an identifier declared in this `GEER` (in this case a procedure identifier); `Object params[]` is an array of objects that this

procedure takes as arguments (note that these objects need to be `Serializable` and must implement an Object `compareTo(Object)` method to compare itself to another object in order to verify if two demands' parameters are the same), context is the context of evaluation of this demand, and [`code`] is the optional code of the procedure, in case we don't want to assume that the worker has the code to be executed.

Resource Demand A demand for a processing resource. DGT will create *resource* Demands for `GEERs` if they receive demands that require certain resources that they are not aware of (i.e., they do not have the `GEER` corresponding to the `GEERid` of this demand in their `Local GEER Pool`). Such a request for this specific `GEERid` is a Resource Demand. Upon reception of the result of such a demand, i.e., the requested demand, the `GEER` resulting object is added to the `Local GEER Pool`. Similarly, when a DWT receives a demand for which it does not have the corresponding `ProcedureClass` to execute, it will create a resource Demand for this `ProcedureClass`. Upon reception of the result of such a demand, i.e., the "processed" demand, the resulting `ProcedureClass` object is added to the `Local Procedure Class Pool`. Note that for simplicity reasons and similarity with the DGT implementation, the DWT might rely on a `GEER Pool` (given the fact that `GEERs` contain their corresponding `ProcedureClasses`). A resource demand has the following form:

{`resourceTypeId`, `resourceId`}

where `resourceTypeId` is an identifier for a resource type, which is an enumerated type now containing GEER and possibly `ProcedureClass` (see the last paragraph). This enumerated type is expandable in order to allow new resource types to be added later. The `resourceId` is the unique identifier for the specific resource instance being sought for by the demander. Any new resource type created must be provided with a unique identifier scheme to identify each specific resource instance of this type.

System Demand *System* demand related to the system's operation. These include monitoring and control of Tiers by a GMT, e.g., by sending monitoring requests such as the state of a Tier, or control requests such as shutting down a Tier. A system demand has the following form:

{`destinationTierId`, `systemDemandTypeId`, `Object params[]`}

where `destinationTierId` is the `Tier Id` of the tier to which this demand is addressed, `systemDemandTypeId` is an identifier for a system demand type, which is an enumerated type containing one element for each kind of system demand, and `Object params[]` is an array of objects that this system demand may take as arguments (note that these objects need to be `Serializable` and must implement an object to compare itself to another object in order to verify if two demands' parameters are the same).

4.4.2 Demand States

Each demand can have three different states during its workflow. Depending on its state, different modules and components respond appropriated actions to migrate demands from its origin to the destination. As shown in Figure 5, when a demand migrates from one execution node to another, it changes its current states in order to be properly identified for the next step. As nodes can identify the kind of task they are going to do with demands according to their current states, e.g., if a demand has a specific state, GIPSY triggers certain procedure.

Pending As shown in Figure 5, the first state a demand has is `pending`. As soon as a `Demand Generator` initiates a demand, it wraps it up with additional information such as demand state, which is pending at the very beginning. This state stays with the demand as demands carry over to the DMS, and later on from `DMS` to `Demand`

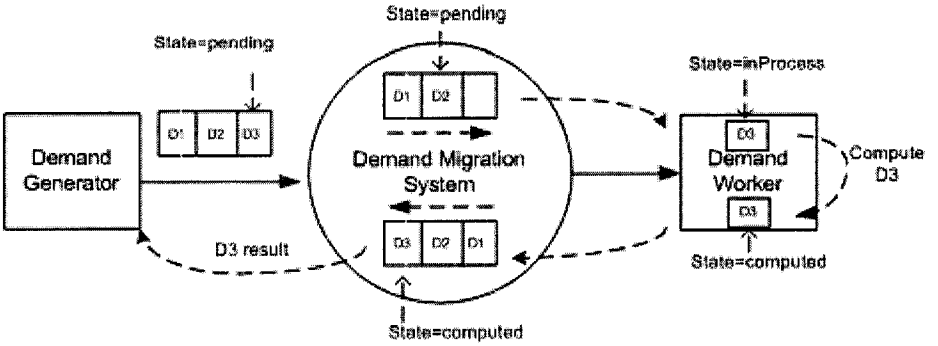

Figure 5: Transition of demand states in GIPSY Demand Migration System

`Worker`.

In Process An *in process* demand is one on its way to be delivered to be computed. The middleware keeps a copy of that demand until the return of its computation results. In that way, the middleware 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.

Computed After computing a demand, the `Demand Worker` changes the state of a demand to `computed`, so it can be returned to the middleware and be differentiated from similar pending demands waiting to be processed. Eventually the `DMS` returns all computed demands to their originating `Demand Generator`.

4.5 Storyboard of the Demand Migration Process in GIPSY

In order to properly comprehend the entire process of demand-migration process in GIPSY environment, here we briefly explain all steps of demand migration. For this

purpose, we provide a simple example to demonstrate the level of granularity a demand may divide into. In addition to that, we explain the interaction between sub modules and components of the DMS. As part of this thesis, we only included those steps from the viewpoint of a `Demand Generator`. It eventually helps the reader understands the general concept of GIPSY demand migration better, and consequently, information in both chapters of design and implementation are easier to understand. As mentioned before, we introduced what a demand means in GIPSY (see section 4.4) and its types (see section 4.4.1). In here, we present a storyboard from the initial point of receiving a demand to the final point of returning its results to the original source step by step.

First, we demonstrate the workflow of our current Demand Migration System. We should clarify in here that even though we provided a dynamic infrastructure for our system for any further modification in the future, this thesis was implemented in a way to follow current DMS and many new features such as GIPSY Multi-tier Architecture have not been introduced while we were implementing our current DMS. However, as part of our investigations and contribution in this research project, we introduced new level of modularity to our system that changed the entire workflow of our DMS. Therefore, we explain here both the current and future ways with the consideration that our implementation is applied only to the current one.

4.5.1 Current System

Currently, we have two types of demands such as procedural and intensional demands. Assuming there is a set of intensional demand in our system, as soon as the `Demand Generator` receives one demand from that list, it puts it in the `Local Demand Pool`. In case the result of such a demand is already present in that local pool, the result would be retrieved from there and returned immediately. Otherwise, the `Demand`

$$\mathcal{A} = \mathcal{B} + \mathcal{C};$$
$$\mathcal{B} = f2(...) + \mathcal{D};$$
$$\mathcal{C} = 2;$$
$$\mathcal{D} = 4;$$

Figure 6: Simplistic Lucid program example.

`Generator` initiates certain processes to compute that demand. In Figure 7, we demonstrate every step of demand migration in the DMS.

Regarding the requirements of a demand, it can be computed either locally or remotely. Each demand requires certain variables or functions values in order to be computed. If all of those requirements are available to the DG computing this demand, it can compute the demand. Otherwise, for the case that any of those resources is not available, it needs to possess it to finish the computation. Therefore, it initiates demands for those required values or resources, while the initial demand is put on hold. After receiving those required resources, it finishes the computation of the original demand and returns the result to the original source. Here is the entire workflow that a demand follows from the initial point to the end. In the presentation of the workflow, we are using the simple Lucid program presented in Figure 6:

1. The DG initially receives a set of intensional demand and places them in its `Local Demand Pool`. For example, an initial demand might be:

$$(\text{GEER1}, \mathcal{A}, [d : 2], null) \qquad (1).$$

2. The DG picks up one demand in its pool, and analyzes its `Demand Signature` to find out which `GEER` it requires. The first element of the demand signature represents the requested GEER (e.g., GEER 1).

3. The DG checks if it has the requested `GEER` (i.e., GEER1) in its `local GEER pool`

4. As part of our basic assumption, DG has the **GEER** (i.e., GEER1) in its `Local GEER Pool` by default, so it starts examining the related AST for the identifier (i.e., $\mathcal{A}$). An identifier may depend on the values of other identifiers or functions or computations. Therefore, the DG should have all their results available in order to have the final value for that identifier (i.e., $\mathcal{A}$), in the context pertaining to the demand being processed.

5. After initial examination of the demanded value's AST in the corresponding **GEER**, i.e., as shown in Figure 8, each identifier may relate to some other identifiers, which means that each variable has some children nodes in its AST. Upon having those children nodes, new demands would be initiated according to the dependencies of the original identifier (i.e., $\mathcal{A}$). For example, as shown in Figure 8, identifier $\mathcal{A}$ relates to both values of $\mathcal{B}$ and $\mathcal{C}$. Thereafter, it generates two demands for the result of each of them, and put them into the `Local Demand Pool` as the following demands, and puts the initial demand (1) on hold until these demands are computed:

$$(\text{GEER1}, \mathcal{B}, [d:2], null) \qquad (2)$$
$$(\text{GEER1}, \mathcal{C}, [d:2], null) \qquad (3)$$

6. In computing demand (2), it is analyzed that $\mathcal{B}$ itself depends on both $f_2()$ and $\mathcal{D}$. Thus, the DG generates new demands at this point, presented below as (4) and (5). Among those demands, $\mathcal{D}$ being a constant, it has its immediate value hard-coded in the AST without any additional computation.

$$(\text{GEER1}, f2, (...), [d:2], null) \qquad (4)$$
$$(\text{GEER1}, \mathcal{D}, [d:2], null) \qquad (5)$$

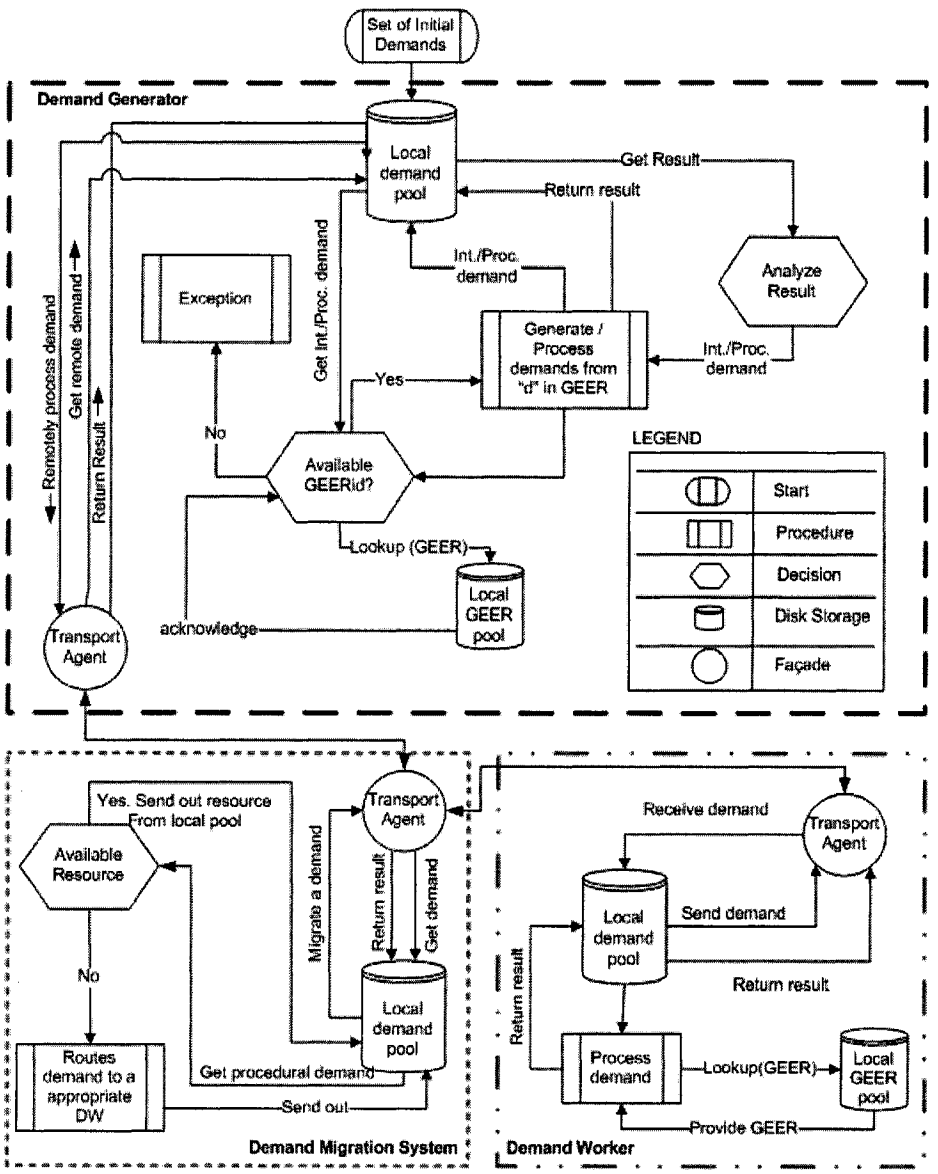

Figure 7: Abstract demand-migration workflow diagram for the current system

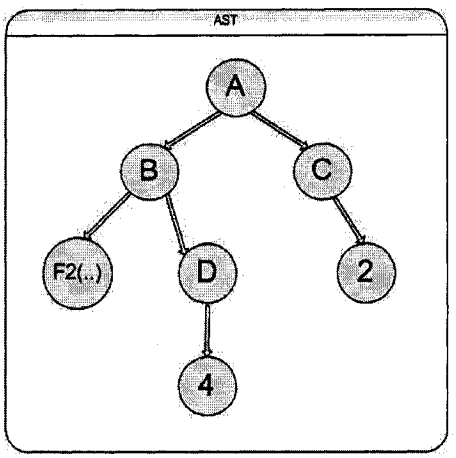

Figure 8: An example of an Abstract Syntax Tree (AST) for identifier $\mathcal{A}$

7. As all these demands are being generated, the `Local Demand Pool` sends a remote request to the DMS through its associated `Transport Agent` to see if these demands have already been computed by other GIPSY nodes and stored in the DMS. In the affirmative, the computed values are retrieved and may be used directly to follow up on the computation. If the negative, these demands are sent to the DMS and might be caught by another DG and processed. Concurrently, the local DG may as well pick up one of these demands and continue generating other demands from it.

8. Procedural demands, (e.g., formula 4), are similarly conveyed to the DMS, with the difference that procedural demands are picked up by Demand Workers.

9. In our current system, we assume that we have all required GEER installed in DGs and DWs `local GEER pool`, so it can process demand without any problem.

10. After finishing the computation of a demand, a DG or DW sends the result back to the DMS through its TA. The DMS attached to this TA then stores the result in its demand pool and notifies all DGs having requested this demand

that the value is available.

4.5.2 Projected Design of DMS

Exactly like for the design of the current system, we assume there is a set of intensional demands. The Demand Generator retrieves one of them and puts it in its `Local Demand Pool`, which reacts by using its associated TA to communicate this demand to the DMS. If the DMS has the result of such a specific demand, it returns it immediately and the DG can proceed, otherwise, the DG starts processing this demand. Here, the process is the same, but the difference is when some processing resources are missing. In the current design, as presented in the last section, if the DG does not have the required resources (i.e., GEER), it is assumed that the demand will be executed in another DG which has this resource available. In our projected design presented here, in addition to the possibility of doing that, the DG can send a demand for those specific unavailable resources, and after receiving that resource and installing it locally, will be able to continue processing the pending demand. In our current system, we do not have resource demand, so we cannot do such a mechanism.

One of the main differences between this model and the current one is in the evaluation of their demand requirements. In the current design, we assume to have required GEER wherever is necessary. It means that both generators and workers have those GEERs they need to execute demands. However, here, we provide a more dynamic infrastructure by introducing *resource demands* to our demand types. As nodes can initiate a *resource* demand for a required resource (e.g., GEER), and they can handle demands no matter if they need additional resources or not. Here, we explain the same example as we did before in our current system with more emphasis on the differences between these two systems. The workflow of this process is depicted in Figure 9. In the following, all steps are assumed to be the same as presented in

the preceding section, and we present the workflow steps involved with processing resource demands.

1. Upon processing a demand, the DG checks if it has the corresponding GEER (e.g., GEER1) in its `Local GEER Pool`. In the affirmative, the workflow proceeds as presented earlier. In the negative, the following additional steps are taken.

2. The DG generates a resource demand for that specific GEER (e.g., GEER1) and puts it in its `Local GEER Pool`. The `Local Demand Pool` then sends the resource demand out through its `Transport Agent` to the associated DMS. A remote worker or generator can catch that demand and provide the DMS with the result (i.e., GEER), and eventually the generator with the requested resource (i.e., GEER1)

3. After receiving the requested resource (e.g., GEER1), the generator installs it locally in its `Local GEER Pool`. After having installed the associated GEER, the generator has the appropriate resource to proceed further in the evaluation of the original demand.

4.6 Summary

In this chapter, we described the design of our Demand Migration Framework and its technology-dependent implementations Demand Migration Systems. We explained the layers and components of the DMS in our multi-tier architecture. Consequently, for the demand side, we explained different types and states of demands. At the end, we studied the demand-migration process from the point view of the Demand Generator for our current and future design. In the next chapter, we provide detailed

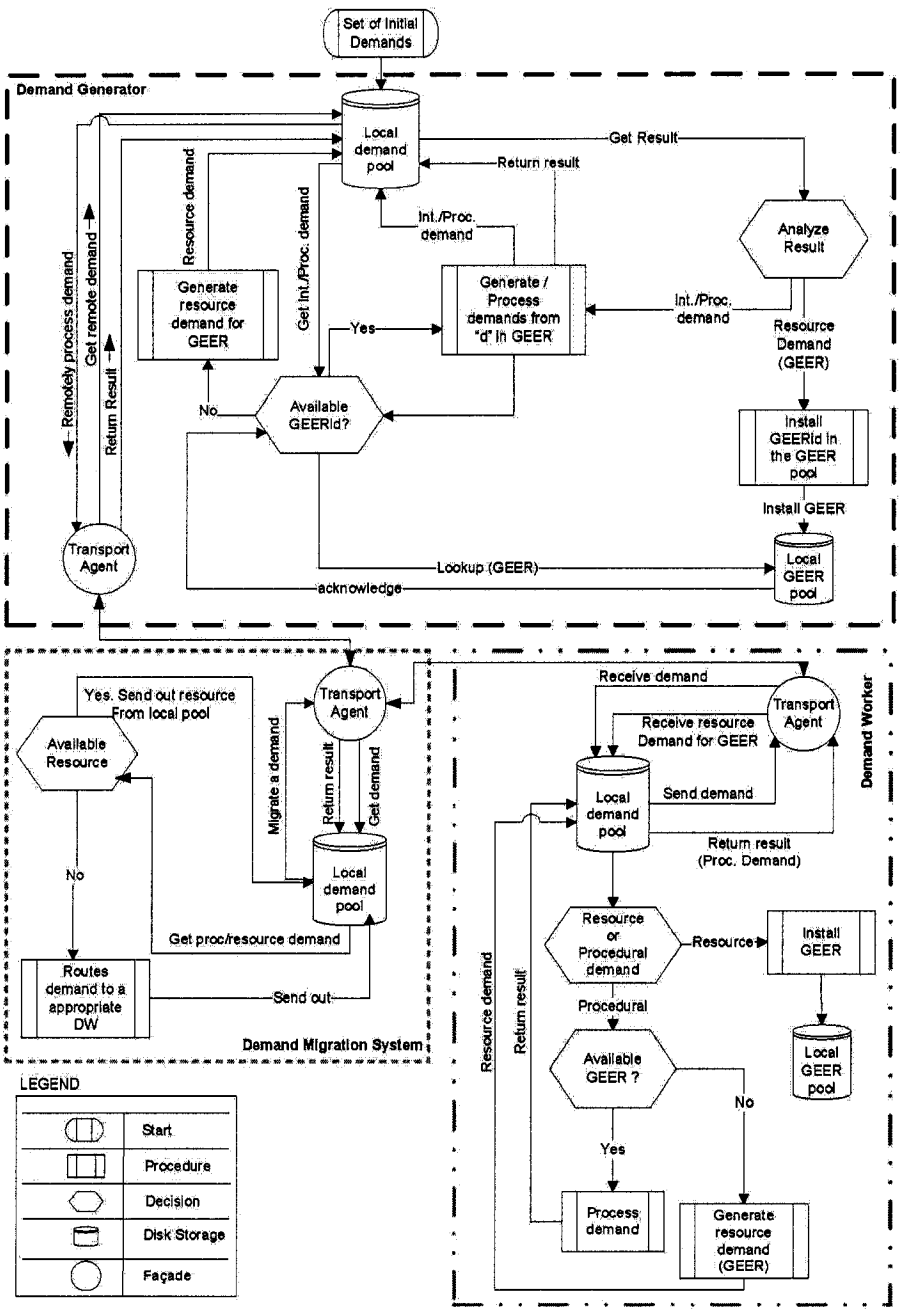

Figure 9: Abstract demand-migration workflow diagram for the future system

information about the implementations of our DMS instances (JINI-DMS and JMS-DMS).

Chapter 5

Implementation

Human beings are members of a whole,
In creation of one essence and soul.
If one member is afflicted with pain, Other members uneasy will remain.
If you have no sympathy for human pain,
The name of human you cannot retain.
Sheikh Saadi - Persian Poet (13th Century)

The GIPSY was designed to include a generic and technology-independent Demand Migration Framework (DMF). Therefore, as our lab also represented our goal and generic design in different publications [14, 15, 40, 41] and a thesis [38], we wanted to implement our DMS with using different distributed technologies such as JINI, JMS, DCOM, .Net Remoting, and CORBA to have different working versions. The reason to have different versions of our DMS were:

- to design a generic DMF, so it can provide an infrastructure to accept different technologies as its separated submodules as long as they follow DMF rationales as presented in 3.1.2.
- to overcome the limitation of binding to only one distributed technology, whereas

in this way, we are open to accept any current or future modification, changes, or introduction of a distributed computing paradigm or middleware. In section 7.2.2, we are going to present how having generic approach can provide an infrastructure for new technologies and paradigms such as Service-Oriented Architecture (SOA) [42].

- to find out the best optimum scenario after experimenting the behavior of our DMS instances, and either choose the best solution or design a new GIPSY-tailored technology to satisfy most of our needs and requirements.

In order to have a technology-independent DMF and to overcome the limitations of using only one technology, we investigated on the implementations of different instances of our GIPSY (i.e., different DMS instances). A DMS has general functionalities which are shown in Figure 10. It consists of dispatching, migration, and getting demands in one side, and on the other side, dispatching, migrating, and finally getting results. Eventually we have considered other functionalities such as demand canceling and pooling.

A DMS uses these distributed technologies as a multi-platform transport protocol, i.e., it is able to connect machines with different operating systems. This transport protocol is based on `Transport Agents` and is open-ended, i.e., we can easily extend it by adding new `Transport Agents` based on other distributed technologies. Note that by having different instances of our DMS, we can benchmark them by performing many scenarios of demand migration in different execution environments, which is one of the goals of this thesis.

The following sections discuss the common architectural design issues in JINI-DMS and JMS-DMS, and later on in Chapter 6, we are going to perform some experimental and comparative studies in our implementations.

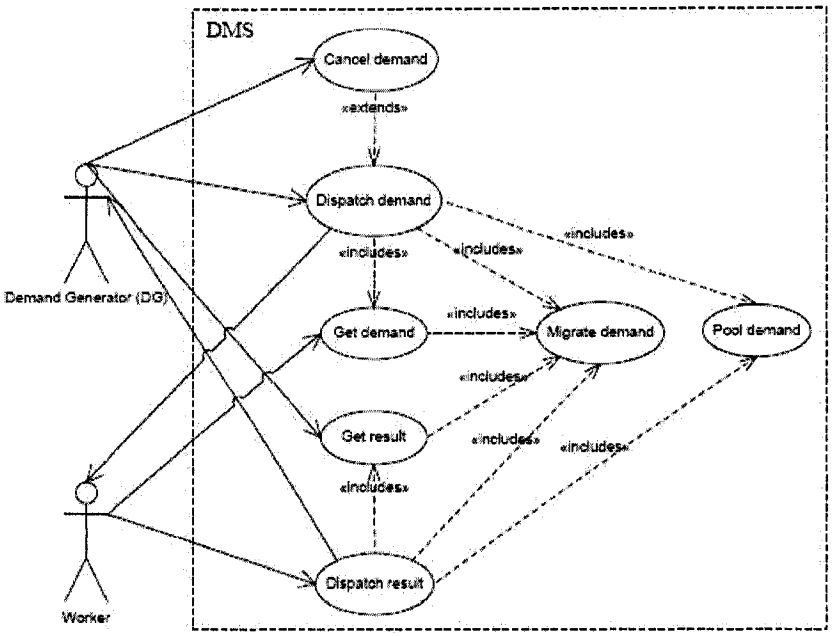

Figure 10: DMS general Use-case diagram

5.1 JINI-DMS

JINI-DMS incorporates a solution based on JINI and JavaSpace [49], where JINI has been used for the design and implementation of the `Transport Agents` and JavaSpace for the design and implementation of the `Demand Store`. The JINI-DMS was the first DMS instance, developed by Emil Vassev in his master thesis [38].

5.1.1 JINI

JINI, developed by Sun Microsystems [1] (later on joined by the Apache group as their River project [19]), is an infrastructure for federating services in a distributed system [60]. As also introduced in [38], JINI provides an open architecture for handling resource components (see 5.1.1.1) — either hardware or software, within a network.

5.1.1.1 JINI Features and Resource Components

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. Here we discuss each of these resource components.

JINI Service According to [60], it is the most important concept within the JINI architecture. A service is an entity that can be used by a user, a program, or another service. A service may be a computation, storage, a communication channel to another user, a software filter, a hardware device, or another user. It basically provides an infrastructure for the Service-Oriented Architecture.

Lookup Mechanism According to [60], JINI services locate each other through specific lookup mechanisms, which can be either Unicast or Multicast. Unicast is for known host, which is the fastest and easiest way as it connects directly. However, in Multicast, the host is unknown and they need to look up in the network to find an appropriate one. After finding the initial service, it connects with as a proxy for further interaction. Multicast mechanism allows establishing an interaction with services without any prior knowledge about their existence or physical address.

According to [44], for services with simple functionality, the proxy, once retrieved from the lookup service, might be able to operate independently — making it a "strictly local" proxy. Therefore, the JINI specification leaves the proxy, protocol, and server to communicate with the service implementation. The service implementation is hidden from the client, which only has to agree on the Java programming language interface, not a specific protocol, to interact with the service. Java Remote Method Invocation (RMI) can conveniently arrange such communication. All JINI services in Sun's implementation of the JINI Specification use RMI as it allows an object to

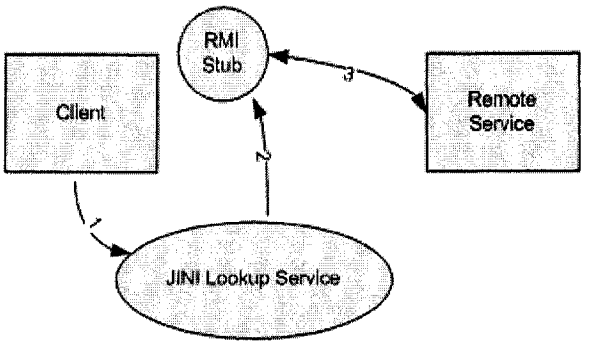

Figure 11: JINI internal lookup mechanism

make its method calls available to objects that reside in other virtual machines, and even on other hosts. As shown in Figure 11, once a RMI stub is available to a JINI client, that client can call methods on the stub. The stub then forwards method invocations to the remote object, and marshals and unmarshals method parameters as needed. In this scenario, a JINI service implemented as an RMI remote object will register its stub as the service object with lookup services [44].

According to [77], JINI overcame the limitations of RMI in terms of multi-casing inside a limited range of network. RMI has a restricted range often within a network subnet, whereas JINI has the concept of federated lookup service to join different subnets and makes the discovery process transparent to clients across a Wide Area Network (WAN).

Communication Event communication in JINI is built on top of synchronous communication. According to [55], method calls in JINI are all synchronous. In order to implement the asynchronous version of it, an event listener is required to get asynchronous messages, but still is based on RMI, which brings some problems, which we discuss in the comparative studies that we have done between different technologies in Chapter 6.

5.1.1.2 Leasing Mechanism

According to [38], access to many of the services in the JINI system environment is lease based. A lease represents the time of validity of a particular entry. If a lease is not refreshed (i.e.,, its life is not extended), it can expire and, consequently, the entry is deleted from the registry. In other words, the host assumes that the queue will be unreachable from that point in time. This may be caused, for example, if a host storing the queue becomes unreachable. A host that initiates a discovery process will find the topics and the queues present in its connected portion of the network in a straightforward manner. Non-exclusive leases allow multiple users to share a resource.

5.1.1.3 JavaSpace

According to [49], JavaSpaces, as a part of the JINI framework, is a network accessible associated shared memory to share, exchange, and store Java objects. It hides the internal details of persistence, distribution, etc., from developers while leaving them free to build distributed data driven applications. According to [17], it integrates the concept of tuple space, where tuples can be inserted, read, and removed from a space which stores each tuple from the time it is inserted to the time it is removed. Tuple is essentially a collection of fields, where each field contains a type value. The corresponding operations are `write`, `read`, `notify`, and `take`. Among these, because of its synchronous nature, both `read` and `take` take a template and block it until a tuple that matches the given template is present in the space.

As Figure 12, borrowed from [38], demonstrates we pass not only data but also object's behavior. 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.

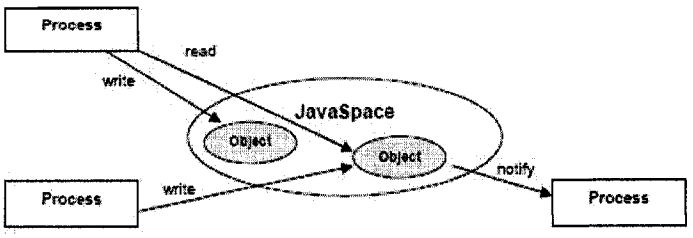

Figure 12: JavaSpace model

5.1.2 JINI-DMS Design

Referring to section 5.1, as represented before in Figure 4, we used JINI and JavaSpace in our first instance of DMS [38]. Due to the fact that we will ultimately compare JINI-DMS and JMS-DMS, we would like to briefly explain the design aspect of the JINI-DMS. We would like to demonstrate the general Sequence Diagram [66] of JINI-DMS from the point of generating a demand in Demand Generator to its migration path across the JINI-DMS and eventually receiving a demand and executing it on the Demand Worker side. Despite the comprehensive and detailed information about all modules and sequence of JINI-DMS in [38], as shown in Figure 13, we would like to demonstrate and describe these steps briefly in order to have a comprehensive study over these subjects. These steps are :

- A DG (Demand Generator) generates a demand (e.g.,, (A)) and passes it to the DP (Demand Proxy).
- The DP grants the demand (A) with a unique ID (UUID) for any further references.
- The DP stores the demand in the DS (Demand Store) and returns the ID to the DG.
- The DG starts listening to the DS for this demand to become computed.

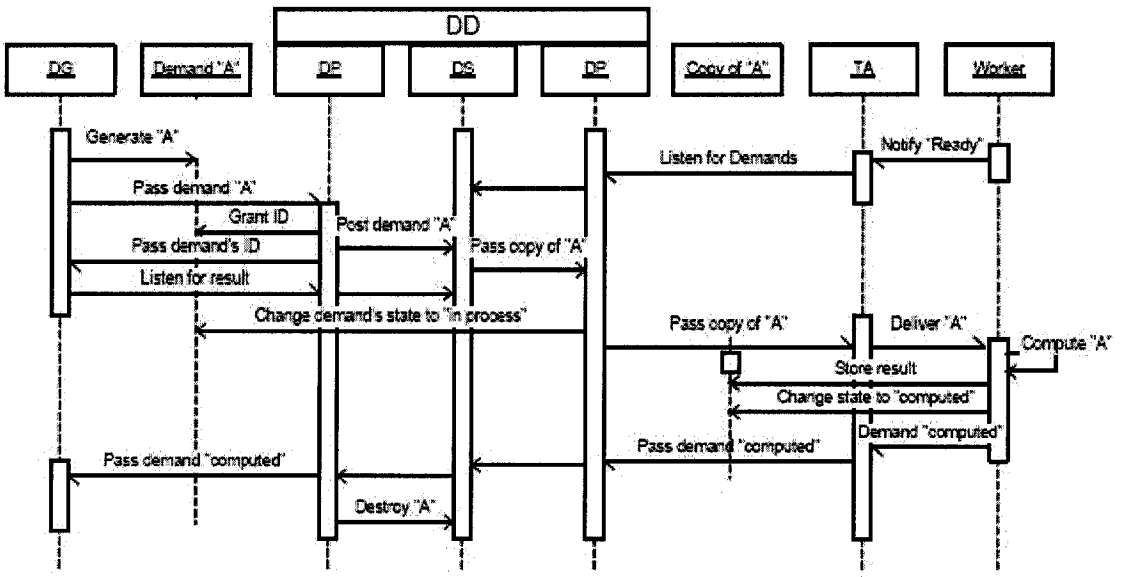

Figure 13: Abstract JINI-DMS general Sequence diagram

- A DP associated with a TA (Transport Agent) 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 (Demand Worker).
- The Worker executes the demand, stores the result in the demand, and changes its state to computed.
- The TA transports the computed demand back to the DD (Demand Dispatcher) 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.

After demonstrating the general sequence of a demand migration, we describe and explain each of these classes and their functionalities:

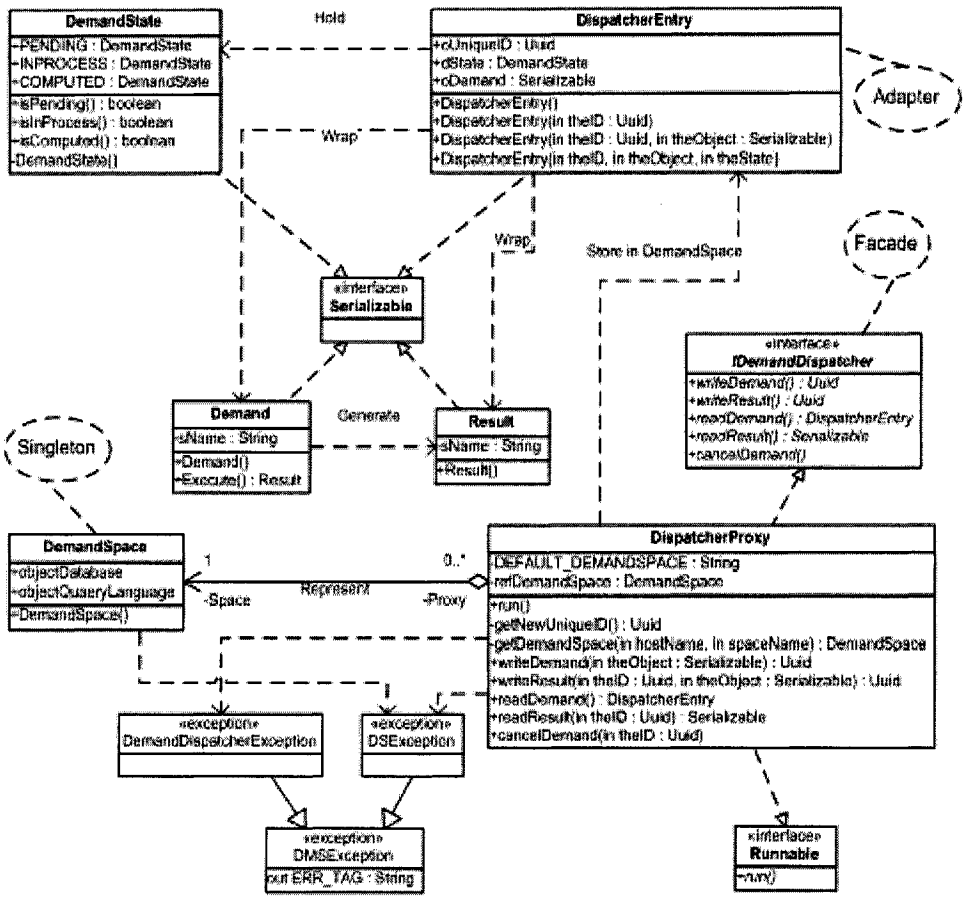

Figure 14: Demand Dispatcher Class diagram

Demand Dispatcher The `Demand Dispatcher` has been designed in a separate package. The following UML Class Diagram [66] (Figure 15) depicts the `Demand Dispatcher` package's classes and their relationships. In here, we would like to demonstrate (see Figure 15) and briefly describe major classes and interfaces inside JINI-DMS class diagram.

- Interface `IDemandDispatcher` is the `Demand Dispatcher` package's entry interface. It exposes the `Demand Dispatcher` package's functionality. All the methods exposed by the interface may throw the `DemandDispatcherException` exception.

- Class `DispatcherProxy` is the design solution to the Presentation Layer. It implements the `IDemandDispatcher` interface, i.e., all the necessary functions for reading, writing and canceling a demand. Moreover, this class implements the Runnable Java Interface [29] (allowing execution in a separate thread of control) to establish a connection, open, and close a session.

- Class `DemandSpace` is the design solution to the `Demand Space`. This class is a Singleton [66], i.e., it can instantiate only one instance. The class implements two major functionalities of a demand storage and a demand query mechanism. Since we did not intend to design and implement an Object Database for DMS, the class has been designed to integrate some already existing Object Databases. Hence, in our design solution the `DemandSpace` class holds two public data fields referencing a demand storage mechanism and a demand query mechanism.

- Class `DispatcherEntry` is the design solution for the problem of having different types of demands as one entry. The class is derived from the Adapter design pattern (see Figure 14). It provides a mechanism to unify different types of demands by wrapping a GIPSY demand into an appropriate format for storing them in the `Demand Space`. The `DispatcherEntry` class implements the `Serializable` interface, which makes the objects instantiated from the persistent class and appropriate for storing in the Demand Space. `DispatcherEntry` holds a demand 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 not be lost in case of any failure or shutdown in any GIPSY Node.

  - Second, we unify different types of GIPSY demands and their associated

results as one generic and wrapped entity.

- Class `Demand` includes all functionalities of a task which is been thought of as a demand. Implementation detail of this class is not important neither for the `Demand Generator` nor for the `Demand Worker`. The implementation of this class is temporary and would be replaced later on as other modules of GIPSY deliver.

JINI TA is a `Transport Agent` that has all the characteristics of a JINI service [49]. Hence, it is a stand-alone component that exposes a common interface to the `Demand Dispatcher` and GIPSY tier instances (i.e., *Demand Generators* and *Demand Workers*) to migrate demands from one node to another. The UML class diagram presented in (Figure 15) depicts the classes and their relationships.

There are two remotely-executing requests or commands as relationship depicted in Figure 15. One is the relationship between the classes `JINITransportAgentProxy` and `JINITransportAgent`. It is an association (see the dashed line in Figure 15) 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 [96], which enforced the design of the `JTABackend` class as an RMI back-end class. The last implements the Java 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 main JINI TA class remotely. The following elements describe the JINI `Transport Agent` package's classes and their attributes and methods.

- The `IJINITransportAgent` interface is the package's entry interface. This interface is in the highest JINI `Transport Agent` abstraction level. Most methods exposed by this interface are functioning on demand migration, i.e., getting or

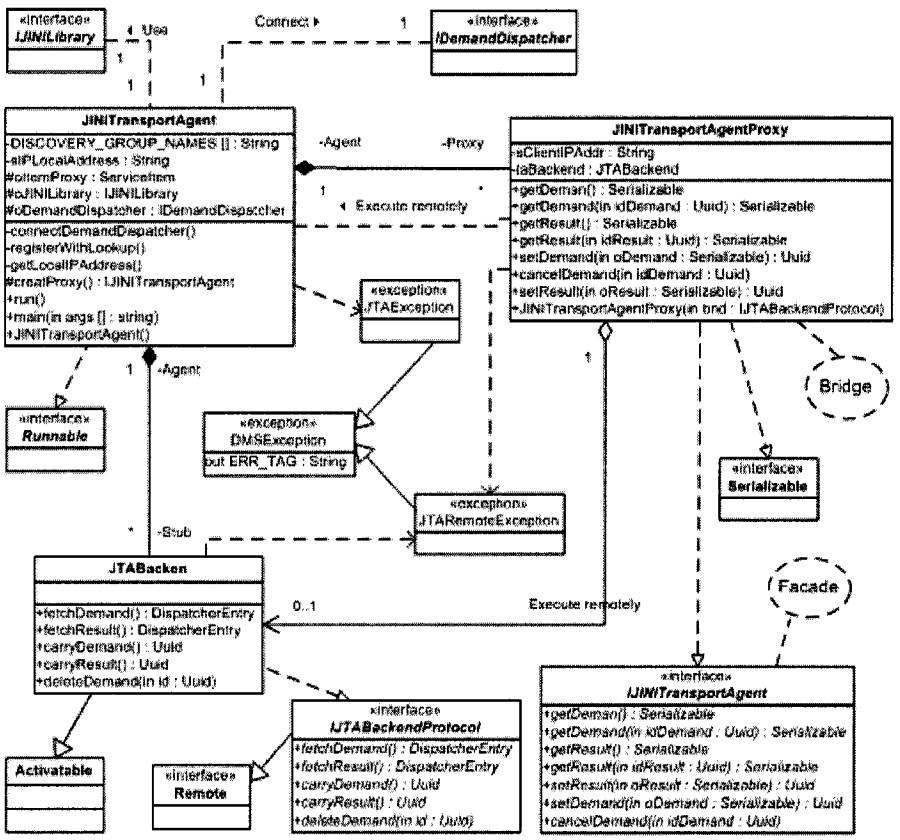

Figure 15: JINI TA Class diagram

setting demands in an interaction with *Demand Space*. This interface is used by the `Demand Workers` and `Demand Generators` first to find the JINI `Transport Agent` within the JINI federation of services, and to call the methods of JINI `Transport Agent`.

- Class `JINITransportAgent` is the main class in this package. It instantiates a standalone object that is the real JINI `Transport Agent`. This class implements both `Transport Agent` and JINI service [49] functionalities. As a JINI service, it will find the lookup service and publish its proxy. It wraps the proxy (see the `JINITransportAgentProxy` class), which is used to implement the interface `IJINITransportAgent` describing the public JINI `Transport Agent` functionality. In addition, class `JINITransportAgent` establishes the connection with the `Demand Dispatcher`. The class implements the Runnable Java Interface (see Figure 15).

- Interface `IJTABackendProtocol` defines the remote communication 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. This interface extends Java RMI Remote interface. All the methods exposed by this interface can be called remotely. Therefore, a proxy with a reference to `IJTABackendProtocol` interface can invoke all the methods exposed by the interface regardless of where `IJTABackendProtocol` implementation physically resides. All the interface's methods throw a `JTARemoteException` exception.

- Class `JTABackend` implements the `IJTABackendProtocol` interface. This class, by generating stubs, which are transported to the GIPSY clients on either side of demand migration, performs the server-side execution or remote objects on the JINI `Transport Agent` functions, i.e., the methods implemented by class

JTABackend run on the JINI Transport Agent side. This is possible due to the fact that JTABackend class inherits the RMI's class Activatable, which makes JTABackend's methods callable from remote Java Virtual Machines (JVM) [46]. The class JTABackend is an inner class, i.e., it has full access to all the attributes and methods of its outer JINITransportAgent class.

- Class JINITransportAgentProxy is the design solution for the JTA proxy. It implements two interfaces — IJINITransportAgent and Serializable. This Serializable interface assures that an instance of the class could be sent to each Demand Generator or Demand Worker attempting to connect to the JTA. The class has a public no-argument constructor, due to its *Serializable* nature [49]. IJINITransportAgent is the JTA interface known by both instances of DGT and DWT. In our design, the JINITransportAgentProxy class is designed as a static, non-public and inner class for the JINITransportAgent class. The argument used here is fine — instances of DGT and DWT gain access to an instance of this proxy at run-time via serialization and code downloading [49], and the inner class has full access to all the attributes and methods of its outer class — JINITransportAgent.

5.2 JMS-DMS

In previous intensional programming systems, messaging was mostly synchronous, static, and highly coupled in the sense that the thread between producer (i.e., Demand Generator) and consumer (i.e., Demand Worker) was blocked until the end of each computation. Except the fact that highly coupling is not desirable in GIPSY DMF, thread blocking is resource consuming in distributed systems especially when there

are many simultaneous connections between nodes. In GIPSY, there is no discrimination in choosing a specific Demand Worker when a Demand Generator initiates a new demand, so it requires a decoupled communication layer in the core of its DMS. According to [48], Message-Oriented Middleware (MOM) is referred to as the most sophisticated enterprise messaging system for decoupled components. By studying different case studies and best practices, we have selected Java Message Service (JMS) [52] to provide us the required infrastructure to implement the demand migration functionalities in the `Transport Agents`.

In the course of this research, we applied the DMF generic design for migrating objects to our design of a DMS based on using JMS paradigm, and built a JMS version of it to transport demands in a heterogeneous and distributed environment, specified by the GIPSY nodes. As shown before in Figure 4, the architectural model of the JMS-DMS follows the layered conceptual architecture implied by the DMF. `Transport Agents` as part of `Migration Layer` and `Demand Dispatcher`, those being subsystems of the JMS-DMS, are all inherited from the DMF. In addition, the `Demand Dispatcher` still consists of two contributors — a `Demand Space` represented with a built-in JMS persistent storage mechanism embedded inside a JMS application server, and multiple `Demand Proxies` representing an interface for accessing this storage. The `Demand Proxy` in this architecture is the standard JMS interface for accessing the messages in the message queue storage.

5.2.1 Java Message Service

Java Message Service (`JMS`) [90] is a part of Sun Microsystems' [1] Java 2 Enterprise Edition [87]; it is a set of interfaces and associated semantics that govern the access to messaging systems. It is a Java API allowing the applications to create, send, receive, and read the messages. A JMS provider is a messaging system that implements the

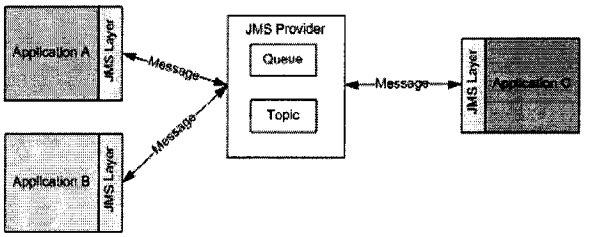

Figure 16: Basic JMS messaging architecture

JMS interfaces and provides administrative and controlling features.

As illustrated in Figure 16, client usually consists of two layers: the application layer and the JMS communication layer. The application layer is implemented to do their own procedure, but when it comes to tasks that require any remote computation, it uses JMS communication layer to communicate with the JMS server and receive the messaging service. The client-side layer is provided by the JMS implementation and manages the client's interaction with the JMS server. As also [65] represented, the JMS specification does not define how the server should be implemented, but rather defines the interfaces and services that the JMS infrastructure must provide.

In JMS only the clients are message producers and consumers, i.e., a JMS server does not produce or consume messages.

5.2.1.1 JMS Architecture

Administrative Objects One (i.e., an administrator) should create administrative objects inside a JMS provider, which are basically `ConnectionFactory` and Destinations (e.g., Queue and Topic). As also explained in [50], it is the same way a database system manages its persistence. Just as an administrator who populates the data schema in the database, one should configure the JMS provider with the JMS destinations and administrative objects by using Java Naming and Directory Interface (JNDI) [91] to define the path of communication between GIPSY modules. These are pre-configured JMS objects created and configured by an administrator

prior any use of a JMS Provider.

Connection Factory According to [61], a Connection Factory is an object a client uses to create a connection with a JMS provider. It encapsulates a set of connection configuration parameters that has been defined by an administrator.

Destinations According to [61], a destination is the object a client uses to specify the target of messages it produces and the source of messages it consumes. In the Point-to-Point messaging domain, destinations are called Queues, whereas in Publish/Subscribe mode, it is Topics.

5.2.1.2 Message Domains

JMS provides two different types of message domains of Point-to-Point and Publish/-Subscribe.

Messaging clients in JMS are called JMS clients, and the messaging system, the MOM, is called the JMS provider. A JMS application is a business system composed of many JMS clients and, generally, one JMS provider. In addition, a JMS client that produces a message is called a producer, while a JMS client that receives a message is called a consumer. A JMS client can be both a producer and a consumer.

JMS uses destinations which can be either queue for the Point-to-Point or topic for Publish/Subscribe. According to [43], destinations can use specific filters such as logical constraints and complex subscription patterns. Since JMS looks up in the application servers and uses the Java Native and Directory Interface (JNDI) API [91] to find the related queues, topics, and connection factories, one should create and configure these objects first. These objects can be either permanent or temporary. Temporary in the sense that they last as long as the lifetime of the connection.

Queue is suitable for single consumer and Topics for multiple consumers. We can

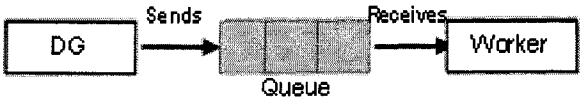

Figure 17: Point-to-Point messaging using JMS queue

also group queues into a cluster of them to improve the scalability.

Point-to-Point As shown Figure 17, in point-to-point messaging, a message (i.e., demands) is sent by a JMS client to a specified message queue, from which it is extracted (received) by another JMS client. Hence, the message sent to a message queue is received by only one client. The receiver acknowledges the successful receipt back to the sender. DWs and/or DGs can share the same queue, still preserving once delivery semantics and providing a broadcast messaging capacity between DGs and DWs.

There are two ways to notify the GIPSY nodes about the destinations that they are going to receive demands from; 1)broadcasting all those names or/and 2)use default names. Either of those, GIPSY nodes looks up those names inside the JMS provider by using JNDI interface.

The second way is the simplest, which we used in our system as names of our queues or topics do not change. However, in more critical situations, we could easily include this feature into our system that DMS notifies all nodes upon creation of a new destination (e.g., a new queue) with additional information such as expiry date and etc. The name with addition information can be set up as an entity in the JNDI registry, which functions similar to the leasing mechanism described in section 5.1.1.2 in JINI.

We use Point-to-Point in most of our cases as each demand needs to be computed once and only once, so our demand migration procedure is all done by using this domain.

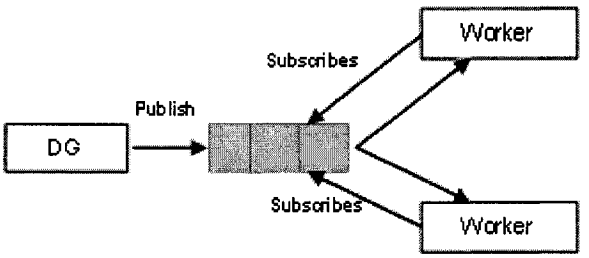

Figure 18: Publish/Subscribe messaging using JMS topic

Publish/Subscribe The other domain (Publisher/Subscriber) is based on the use of Topics. Publisher (i.e., Demand Generator) and subscribers (i.e., Demand Worker) are generally anonymous and may dynamically publish or subscribe to a specific content of a Topic, but as shown in Figure 18, opposite to the other domain, it has multiple consumers. Demand Workers are interested in some sort of demands or events, so by subscribing to those specific topics, they will be notified as soon as such a type of demand arrives on that topic. It means that when a Demand Generator sends a message, any subscriber who is subscribed to that interesting content, will receive it. Therefore, we would face high network traffic and multiple computation, which necessitates DMS to control every single computation. Therefore, a `Demand Worker` computes the request, notifies the DMS, and subsequently, it requests all other nodes not to compute that specific demand. This makes the implementation quite complicated, in addition to the fact that multi-computation is not desirable in GIPSY. Hence, we only work with the Point-to-Point messaging for most of our cases.

The only time we would like to use Publish/Subscribe is when we want to broadcast a new configuration (e.g., a new name of a queue) to other remote execution nodes.

5.2.1.3 JMS Provider

As mentioned in [90], JMS provider, acting as the central part, leverages all administrative, functional, an control capabilities of the JMS messaging. Figure 19 simply

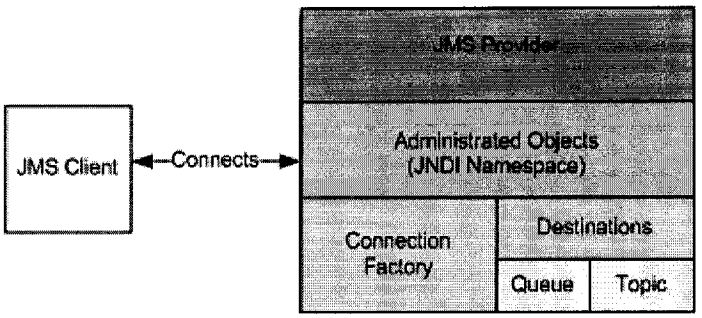

Figure 19: JMS API architecture

shows how JNDI functions in order to establishes a connection. JNDI is a naming service to provides a way to look up objects or references in a JMS provider by their names. Administrative tools allow one to bind destinations and connection factories into a JNDI namespace. A JMS client can then look up the administered objects in the namespace and then establish a logical connection to the same objects through the JMS provider.

According to [50], computers and networks that connect execution nodes together are inherently unreliable in the sense that either side of these interactions may be down at any time while other one is dispatching new demands. These failures are either internal or external in their network connections. Therefore, there it comes the necessitation of a thrid-party entity to overcome this limitation by repeatedly trying to correctly transit data packets toward their appropriate destinations until it succeeds. As also shown in figure Figure 20 (borrowed from [50]), we have different steps of message transition from one computer to another by using a JMS provider.

5.2.1.4 Choosing among available implementations of JMS

There are many commercial and open-source JMS providers, which we list below.

Open-Source Projects

- Jboss Application Server from Jboss Inc. [7] (Formerly part of discontinued

JbossMQ)

- ActiveMQ by Apache [31]
- Joram by ObjectWeb Community [8]
- OpenJMS by the OpenJMS Group [75]
- HermesJMS by Hermes Community [33]
- MantaRay by Coridan Inc. [9]
- Pronto [98] by University of Cambridge (specifically for mobile environment)

Commercial Projects

- GlassFish Application Server [12] by Sun Microsystems [1] (formerly part of Sun application server personal edition)
- WebLogic Server [5] by Oracle Corporation [2] (formerly by BEA [3])
- Websphere MQ [13] from IBM [6]
- TIBCO Rendezvous [11]
- Oralce AQ [10] by Oracle Corporation [2]
- Microsoft Message Queuing [25] by Microsoft Corporation [4]

Among JMS Application Servers, MSMQ [25] supports only Point-to-Point, whereas Jboss [7] among open sources ones, and Sun GlassFish [12], TIBCO Rendezvous [88] support both Point-to-Point and Publish/Subscribe domains.

Given the list of some possible commercial and open-source JMS Provider among which, we have chosen Sun application server Personal Edition 8.2 to commence our research investigations. We chose this provider, as it was free and easy to configure.

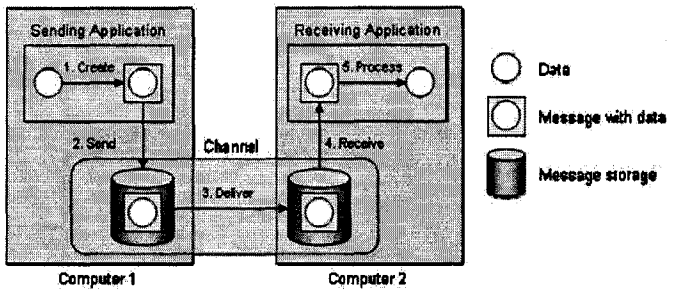

Figure 20: Message transition by using a JMS provider

However, in the middle of our implementations, we found out that the personal edition supports only two or less concurrent message consumption at the time, and unlimited consumption is only provided in their Enterprise Edition (EE) which was a very expensive scenario for us. Therefore, we chose another successful open-source provider (i.e., JbossMQ) to pursue our research. After facing a few bugs in the JbossMQ, and communicating with the Jboss community about those issues, they notified us that JbossMQ is discontinued and the best solution would be the latest version of Jboss Application Server as they embedded all functionalities of previous versions in it. Therefore, we simply configured all of our JMS administrative objects in it.

As the central part of the architecture is the JMS server, it generally acts as a hub for all communications, and has access to stable storage. The clients communicate by exchanging messages which are relayed by the server. Replications of our JMS provider would eventually improve the fault tolerance of the entire system.

5.2.2 JMS-DMS Design

In order to properly comprehend the DMS architecture, different figures of component (see Figure 21) and class diagram (see Figure 22) are provided. In the component diagram, two main generic layers of `Transport Agent` and `Demand Dispatcher` are separately shown. The Migration Layer is shown as the first and proper interface

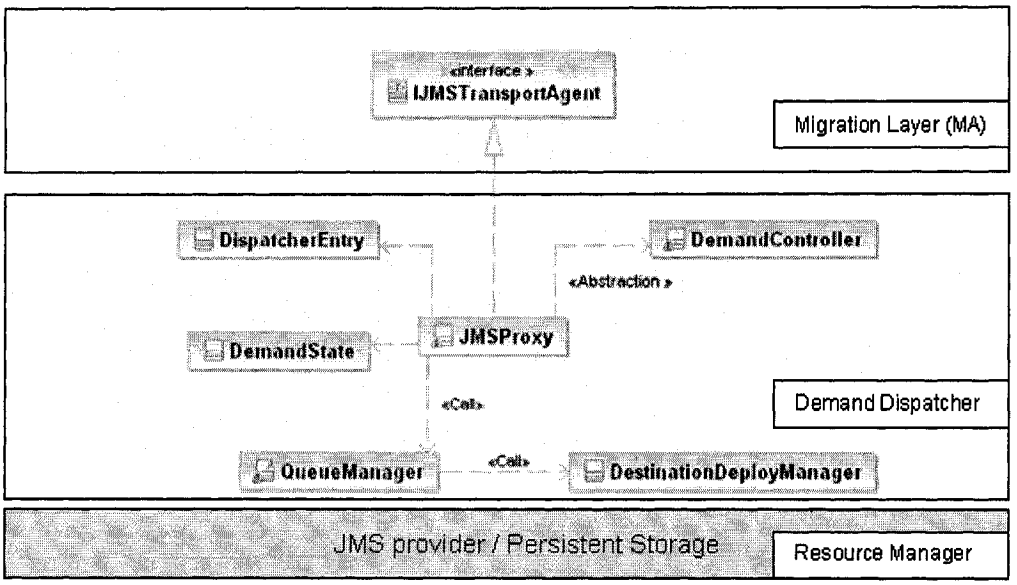

Figure 21: JMS-DMS Component diagrams

`IJMSTransportAgent` that any GIPSY Node can implement to interact with other nodes or tiers. `JMSProxy` in the `Demand Dispatcher` follows a Facade design pattern to implement this interface comprising all the functionalities to start and stop a connection, writing a demand into a destination, receiving a demand from a destination, and other required functionalities for the demand migration. In addition to `JMSProxy`, this layer has `DemandController`, `DispatcherEntry`, and finally `DemandState` classes which are all going to be explained in below.

As previously stated, an administrator should configure JMS administrative object inside JMS provider prior to any use of JMS provider. However, this can also be done through Resource Manager, which registers and configures all those objects automatically. We would like to use this class mostly as our startup configuration to improve the portability of DMS. Moreover, it gives the option of creating and configuring any temporary/permanent destination to either sides of the demand migration procedure.

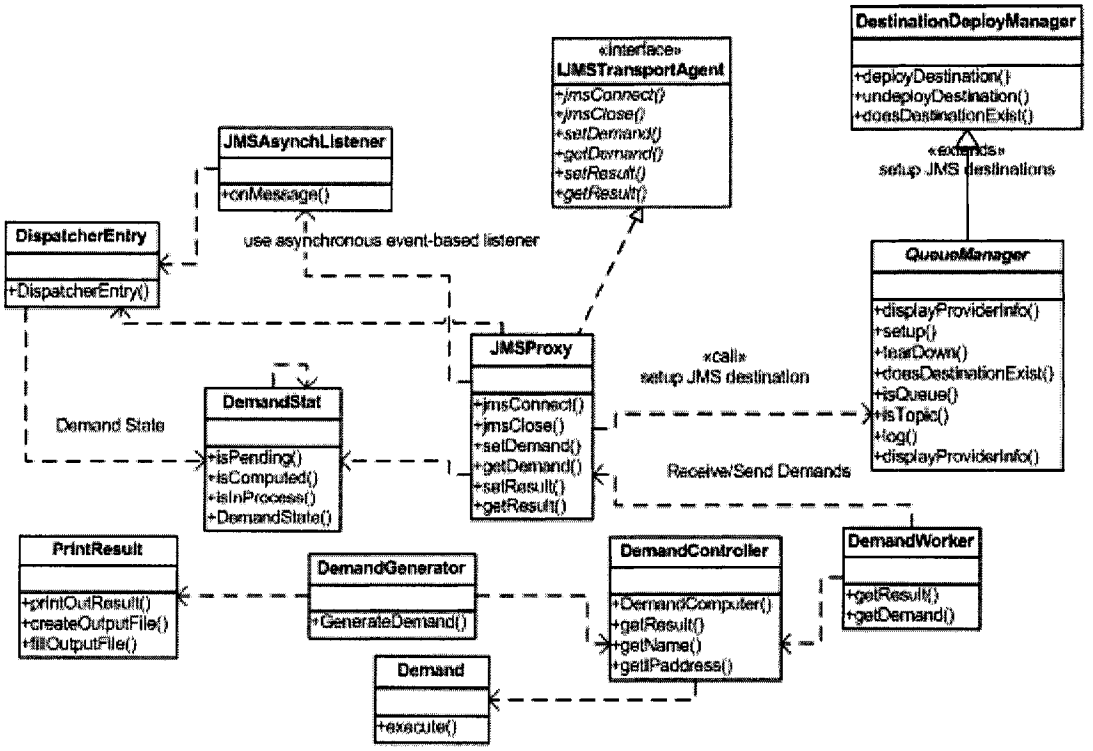

Figure 22: JMS-DMS Class diagram

The component diagram presented in Figure 21 clarifies the DMS's generic conceptual view, but in order to point out detail information about functionalities of the middleware in an extensive approach, we explain below the JMS-DMS class diagram as shown in Figure 22. In the following, the JMS-DMS package's classes and their attributes and methods explained.

- Class `JMSProxy` plays a major role in the DMS. DMS is the combination of two layers of `Migration Layer` and `Demand Dispatcher` layer. As discussed, the `Migration Layer` (i.e., *Transport Agents*) exposes those functionalities that are implemented in the `JMSProxy`.

  This facade is the most important class which implements methods such as `JMSConnect`, `JMSClose`, `setDemand`, `setResult`, `getDemand`, and `getResult`. The `Demand Generator` connects to DMS by calling `JMSConnect`. In order to

migrate demand from one GIPSY Node to another, firstly, we need to write a demand into a queue by calling (`setDemand`). On the other side, after connecting a `Demand Worker` to the DMS, it calls `getDemand` to receive a demand from a queue. Finally, `JMSClose` closes the connection. These `Transport Agents` main functionalities are all exposed as the public interface for instances of `Demand Generator` and `Demand Worker` across the network. Figure 22 shows the case that a `Demand Worker` is connected to the DMS waiting for any available demands to compute.

- Class `DemandController` wraps itself around a demand to carry over additional information such as execution arguments in order to execute the task and deliver its result to the original `Demand Generator` accurately.

- Class `DemandState` is the 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 [14].

- Class `DispatcherEntry` converts the interface of a class into another interface which clients on the other side expect, i.e., by applying Adapter design pattern, it wraps the demands and results in an entry format. The demands unification surely brings simplification to make demand storing, retrieving, and querying consistent.

- Class `JMSAsynchListener` implements `MessageListener`. This listener has been used as an asynchronous event handler. Those messages sent persistently will be stored until delivered by the messaging system. This guarantees that

the message will not be lost due to a crash or shutdown of DMS [14].

- Classes `QueueManager` and `DestinationDeployManager` provide the facility of adding or removing a user-defined destination at run-time. Therefore, all the necessary functionalities to add, remove, or modify a JNDI element in the application server are implemented in them. Upon request of registering a new destination, these classes register them temporary or permanently inside the Jboss Application Server.

- Class `Demand` includes all functionalities of a task which is been thought of as a demand. Detail implementation of this class is not important neither for the `Demand Generator` nor for the `Demand Worker`. As also mentioned before in JINI-DMS, the implementation of this class is temporary and would be replaced later on as other modules of GIPSY deliver.

After describing all main classes, we would like to represent the sequence diagram related to our demand migration in JMS-DMS. As we also highlighted, through a frame in the center of the system-sequence diagram in Figure 23, the most important functionality of the DMS is on the shoulder of JMS-DMS. It resides in the center of the system sequence diagram, and receives the demands from `Demand Generator` and connects to JMS provider to migrate demands from one node to another. JMS-DMS states for our JMS TA including main functionalities such as connect, close, write, and read, and exposing them as the public interface to `Demand Generator` and `Demand Worker` across the network.

However, in Figure 24, we demonstrate the synchronous demand migration mechanism across the system with more comprehensive detail. Figure 24 shows the message-passing mechanism from the first step in demand generation to the demand execution on the other side. JMS-DMS itself includes `JMSProxy` to connect interface with

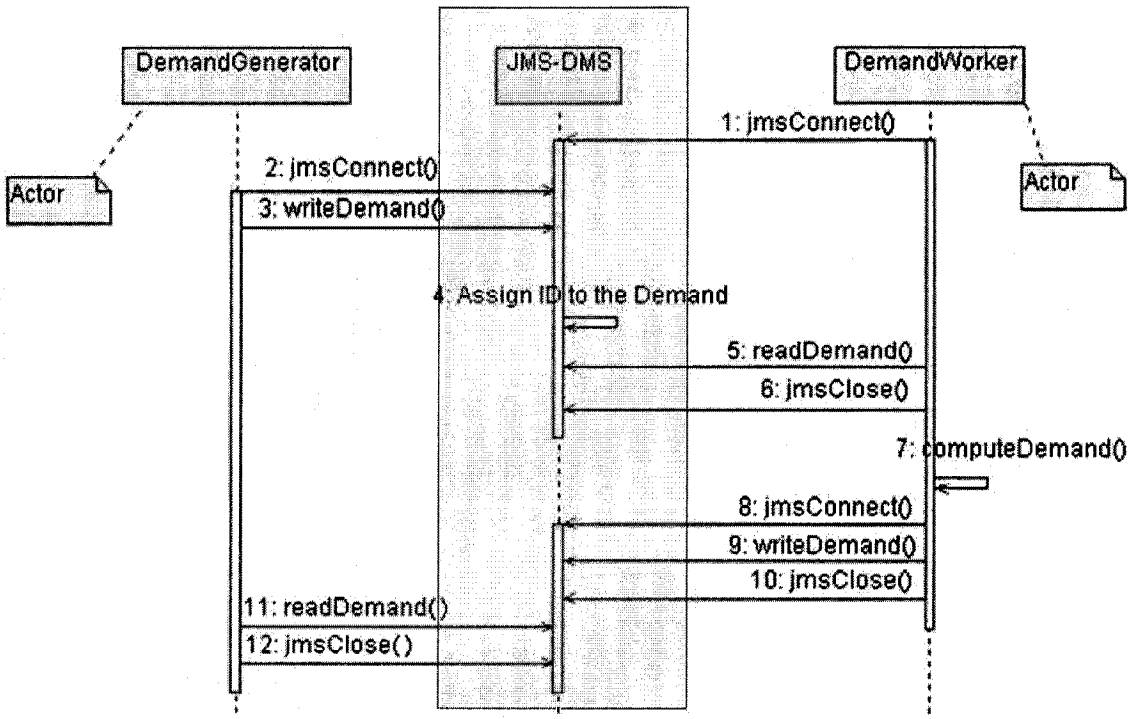

Figure 23: JMS-DMS simplified system Sequence diagram

JMS provider, and `DisptacherEntry` and `Demand Controller` to wrap the demands. `JMSProxy` includes information how to connect to the JMS application server, and it connects to the appropriate JMS connection factory to write the request on the related destinations, which can be a queue or a topic. In order to have a simplified diagram, we only demonstrate the synchronous mode of transaction in this paper.

- `Demand Generator` connects to the connection factory of the `JMSProxy` by calling the `jmsConnect` method (1),
- In order to send the demand to a destination it should call the `writeDemand(demand)` method (2), and pass the generated GIPSY demand. Consequently, `DispatcherEntry` wraps the demand,
- The `DispatcherEntry` encapsulates the demand in the `Demand Controller` with some additional information like the `DG` location, unique identification,

demand state, which is `pending` when it is not processed yet, and dispatches and wraps (3, 4) the encapsulated `Demand Controller` to the destination,

- The universally unique identifier (`UUID`) used for demand tracing will be generated (5) for each demand,
- The generated UUID sent backward first to the `JMSProxy` (6),
- And in order to track the demand and its result later on, the `UUID` will be sent to the `DG` (7),
- When the `DW` becomes available it connects (8) via the `JMSProxy` to JMS,
- And reads (9) the first available demand. Depending on the consumption model, the `DW` will act differently:
- If the mode is synchronous, the `DW` gets the demand directly.
- If the mode is asynchronous, the `DW` would use an event-based listener to capture the demand first and sends it to the `DW`.
- After receiving the request, the state of the demand will be changed to `in process` until the `DW` finishes its computation,
- When the DW receives the wrapped demand (10),
- First, it closes the connection temporarily (11),
- Then proceeds with the computations (12) of the demand,
- And the result of the computation will be sent to the `DW` side (13),
- At this point, the demand state will be changed to `computed`.

- In order to migrate again to the other side, it should be wrapped the same way that it was at the beginning (14, 15, 16),
- The DW connects (17) again to the `JMSProxy`,
- And writes the result back to the destination (18),
- and closes the connection (19) at the last step.
- `JMSProxy` finally receives the result from the destination (20),
- And sends it back to the `DG` side (21).
- Next, it closes the connection (22).

5.2.3 Message Communication Mode

One of the major significances of JMS messaging lies in the capability to decouple applications while they share information between themselves [35], contrast to some other messaging models such as RPC or RMI, which are static and synchronous only. This means that JMS provides both communication modes of synchronous and asynchronous to our environment.

Synchronous It tightly couples the process or message sending and receiving, so it blocks their means of communication (e.g., communication thread) until the sender receives its requested result. Even though it is quite fast in normal cases, it comes quite expensive in terms of resource demanding. As both of their communication nodes should be present in an available network infrastructure, the thread becomes very expensive when every single demand requests and blocks a thread. The more nodes send demands, the more resources would be requested and blocked. Depending on a scenario, we can have this feature enabled in our demand migration. By applying

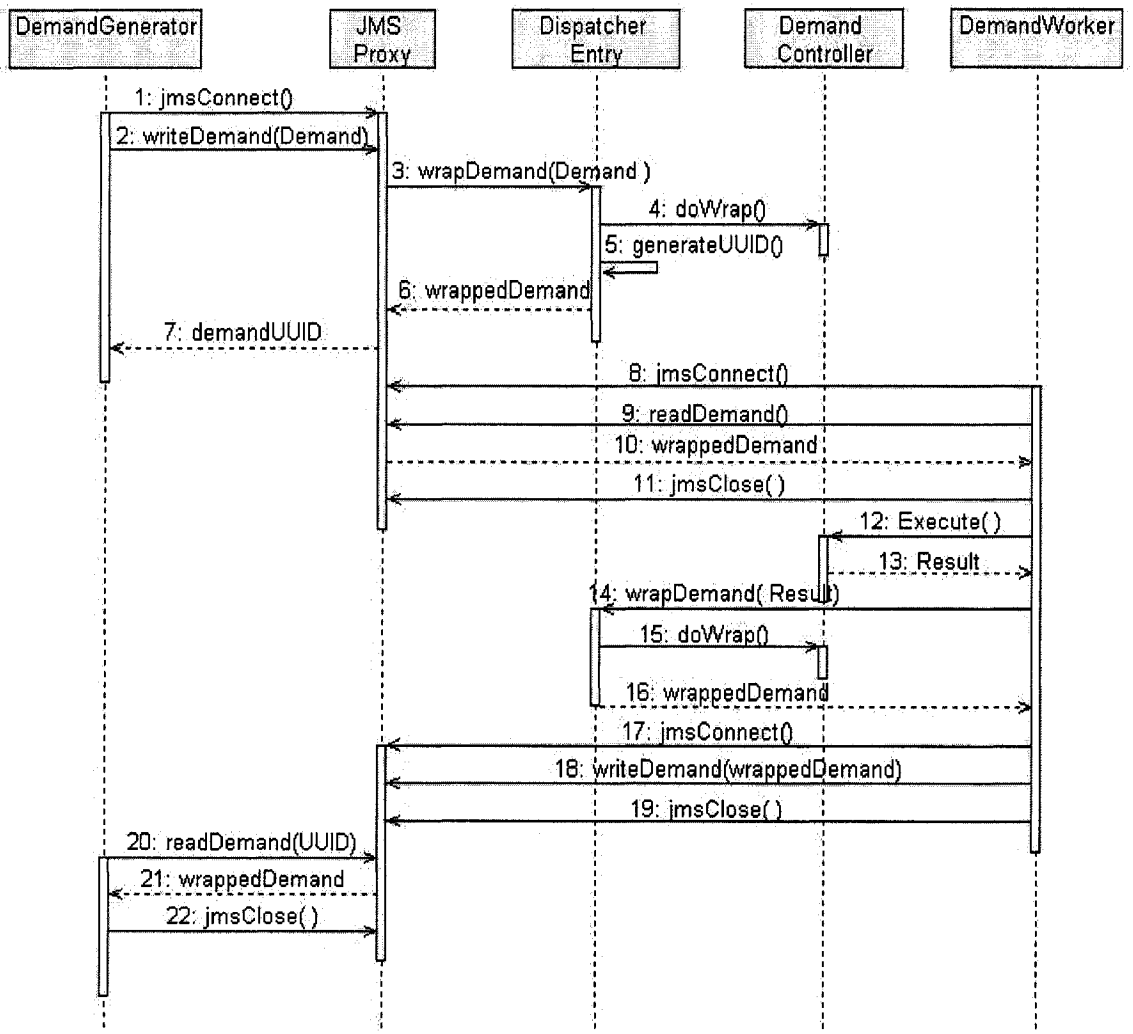

Figure 24: JMS-DMS simplified internal sequence diagram for synchronous mode

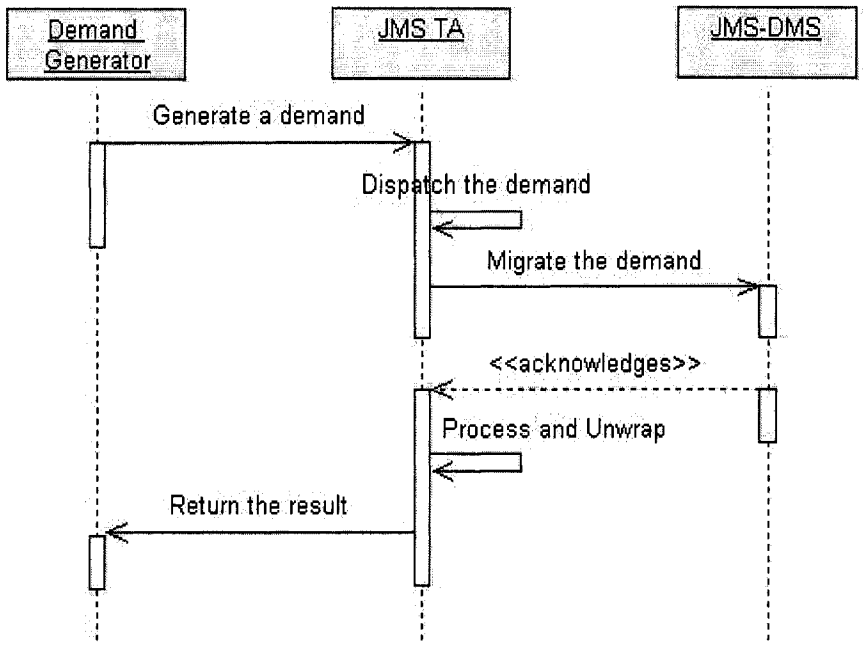

Figure 25: JMS-DMS simplified synchronous siagram

the synchronous diagram described in [50] in our infrastructure, we represent the synchronous communication mode in Figure 25. It simply shows the blocking mechanism on the Demand-Generator side until it receives the appropriated response.

Asynchronous It is the first step toward having a loosely-coupled system. It uses the "store and forward" mechanism while a node sends out a message toward a receiver. In this mode, none of these sides block a resource or thread for their demand-migration process. JMS uses Message Listener, which is an event-based listener to inform each of those sides for a new demand in its JMS destinations (e.g., Queues or Topics). It implements the `MessageListener` interface, which contains one method, `onMessage`. In the `onMessage` method, we define the actions to be taken when a message arrives. Once message delivery begins, the message consumer automatically calls

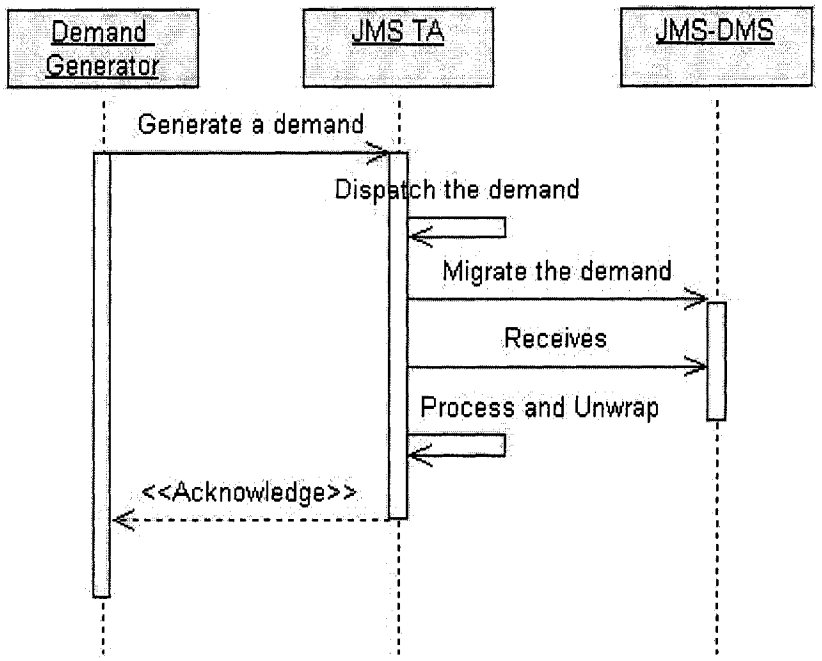

Figure 26: JMS-DMS simplified asynchronous diagram

the message listener's `onMessage` method whenever a message is delivered. Therefore, those nodes remain loosely-coupled to the messaging structure and any failure in those nodes does not affect the whole infrastructure. Depending on a scenario, we can use this enabled feature in our demand migration, but because of having powerful and loosely-coupled system, we favor asynchronous communication mode to the synchronous one. Again by applying the asynchronous diagram described in [50] in our infrastructure, we represent the asynchronous communication mode in Figure 26. It shows that as soon as the message has been sent out, Demand Generator continues on working other tasks such as generating more demands. As soon as the response arrives, the TA receives it, and returns it to the original Demand Generator.

5.2.4 Subscription Mode

Durable According to [90], indeed, the client is ensured to receive all messages that have been published to the topic it has subscribed to, even if its connection is not permanently active. During the periods when a client with durable subscription is not connected, JMS provider keeps the messages for it and dispatches them as soon as the client subscribes again.

Non-Durable According to [90], with a non-durable subscription the client receives messages published to the topic as long as its connection to the server is active. The connection can break (i.e.,, become inactive), for example because of a link failure, or because of the crash of the client. Messages published after the connection is broken are not guaranteed to be received by the client

5.2.5 Jboss Application Server

As also stated in our research path in section 5.2.1.3, we use Jboss Application Server as our JMS provider. According to their official documentation [32], it is a free, portable, and open-source certified Java platform for developing and deploying enterprise applications, JBoss Application Server supports both traditional APIs and Java EE APIs and includes improved performance and scalability through fine grained replication and load balancing. Even though using such a centralized node would eventually act as a bottleneck for the performance of the system, by installing clusters of this application server across our nodes, we can balance the workload across all our nodes.

Figure 27, borrowed from Jboss official documentation [72], clearly shows major internal components and features of Jboss Application Server, and also in here we would like to mention main reasons why we consider this application server as our

JMS provider.

- Open Source and Free,
- Full J2EE 1.4 supports (e.g., JMS, Enterprise Java Beans, Message-Driven Bean, etc.),
- An infrastructure for Service-Oriented Architecture (We will explain why we need such an infrastructure for our long-term research path (see section 7.2.2)),
- Technology-independent interoperability and cross platform,
- Provides transparent persistency with its embedded database and integrations with Object-Relational Mapping such as Hibernate (We will mention why we need such a standard mapping in chapter 7 (see section 7.2.1)),
- Provides a user-friendly graphical-user interface (GUI) for the administrative tasks such as configuring JMS Destinations (e.g., Queues or Topics),
- JBoss Eclipse IDE for easier development access,
- Supports Transaction by using JGroup [24] as a reliable multi-cast communication toolkit.

5.2.6 Messaging in JMS-DMS

JMS strengthens messaging mechanism inside JMS-DMS by considering messages as lightweight entities that consist of different parts of header, property, and body. According to [35], each message is a self-describing and contains all necessary contexts to allow the recipients carrying out their work independently. Here, we describe how these three parts of a JMS message carry sufficient information in JMS-DMS. Carrying over required information through GIPSY messages leverages the seamless

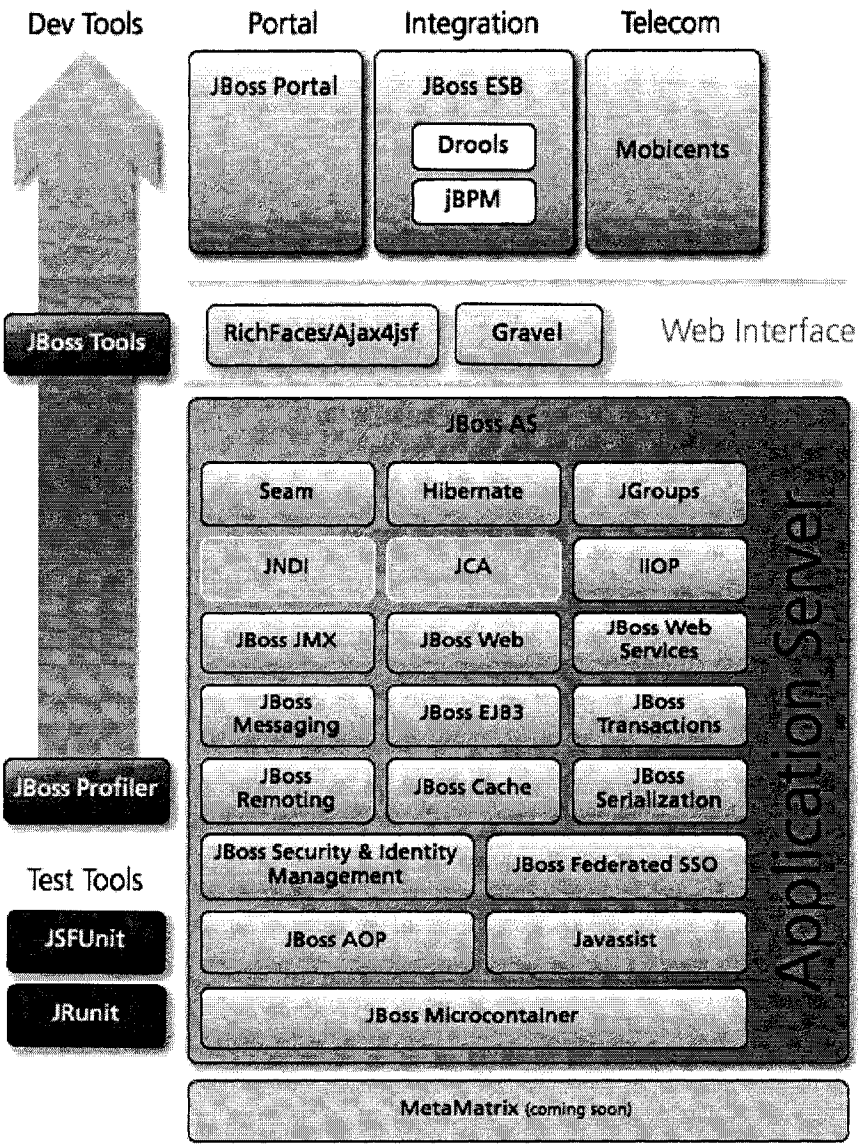

Figure 27: Jboss internal components and architecture

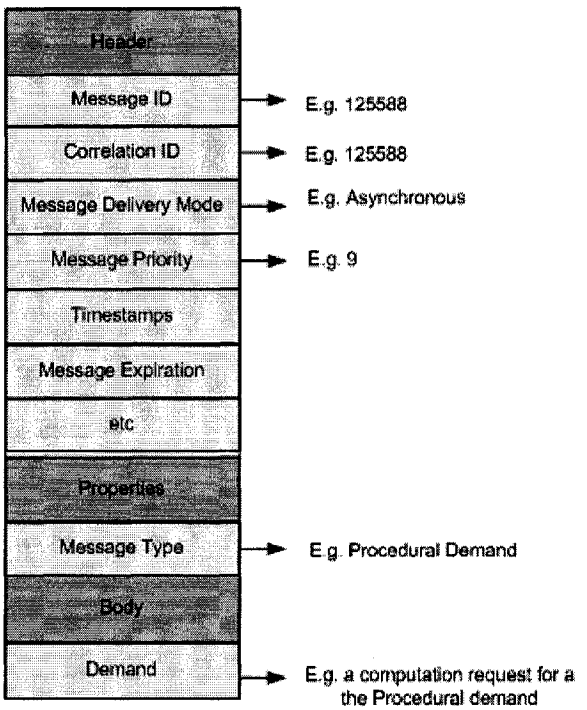

Figure 28: An example of a JMS message inside JMS-DMS

integration of information and services between GIPSY execution nodes. In Figure 28, we will demonstrate each of these parts, and provide an example to facilitate the comprehension of this aspect.

Header comprises of standard system-defined parameters, which are mostly common in all JMS provider. They contain information related to the message routing and identification [90]. The header contains variables for delivery mode, message expiration, priority, signature, timestamp, correlation ID, etc. As we would like to have a transparent system, we would like to use these variables as much as possible as it does not require any implementation in our application-end. As also represented in [14], we set variables to prioritize our messages (i.e., 0 as the lowest and 9 as the highest), timestamp to contain information about the time of dispatching a demand for our benchmarking measurements, signature ID for our unique identification across the system, correlation ID for verifying identification of the returning result and the

original demand, etc.

A demand has a set of user-defined variables in the demand header, which can be set at the point of initiation by the DG. As DMS receives the demand, it checks the message header and decides in which channel or queue it should be routed. Advertisement is a prospective feature that enables the capability to declare the type of event or notification `Demand Generator` is willing to generate.

Property In contrast to header, these are vendor-specific variables (i.e., GIPSY) in JMS messages. It defines detailed information about the demands. These kinds of properties help the system inquiry running messages in the system. Using an SQL-like query language in subject-based filtering can enforce message selection and redirection to improve the load balancing and availability of the entire system.

Body This part contains the functionality of the demand which can be in any JMS message format (e.g., object, stream, text, etc.). GIPSY generally uses object format which accepts any *Serializable* Java object. This makes the distributed demand-driven computing easier, as any kind of objects can store in the body of a GIPSY message.

5.2.7 Persistency in JMS

A JMS message can be either persistent or non-persistent. By enabling the persistency feature, JMS provides a higher level of reliability into the messaging than the non-persistent mode. However, it adds more overhead to the system as DMS should store every single message in order not to lose in any case of failure. According to [86], persistent messages are guaranteed to be delivered exactly-once, whereas delivery for non-persistent messages is at-most-once. In chapter 6, we have evaluated this overhead by measuring the size of each demand in both of these approaches, and we

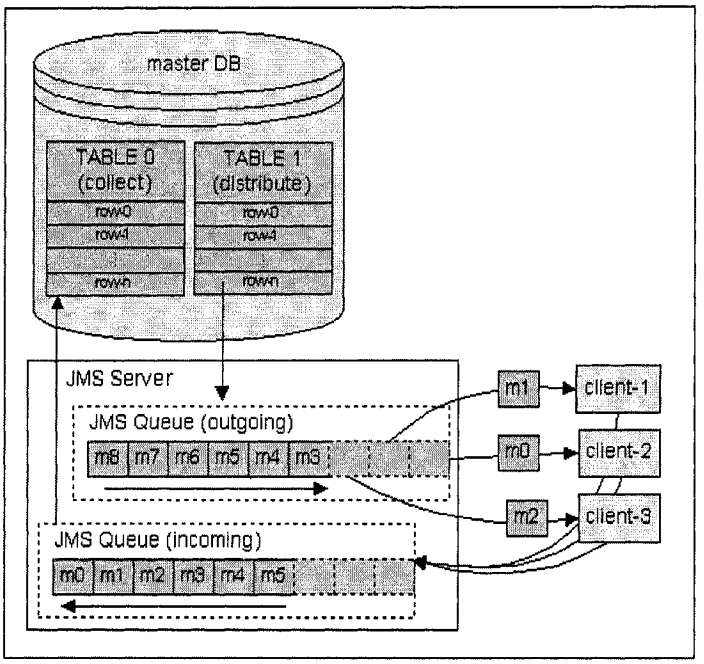

Figure 29: JMS-DMS persistency mechanism

noticed that persistent message is bigger in terms of size than non-persistent ones.

Hypersonic Database As we require a database in order to implement the persistency in JMS-DMS, we used the Hypersonic Database (HSQLDB) [93], which is an embedded solution inside Jboss Application Server kit to provide persistency and caching. According to [93], it is the leading SQL relational database engine written in Java. Jboss embedded this solution inside its package in order to facilities working with its application server "out of the box". However, we faced many problems while working with this database, which are mainly discussed in chapter 6. Figure 29, borrow from [93], shows how HSQLDB deals with incoming and outgoing messages in terms of storing them in its master database.

5.2.8 Demand Delivery Order

Generally speaking, JMS uses First-In/First-Out (FIFO) delivery order. It means that a message, which comes first, delivers first, but technically speaking, this assumption might not be always true in cases of an asynchronous communication mode, prioritized messaging, or even occurrence of any unexpected exception or failure, because each of these cases changes the order a message delivers. For example, if it is asynchronous and some failure happens on the way of a message to the target node, DMS stores demands until problem solved, but at the same time, there may be other message before this one at the point of initiation, where they are following different available routes, so they may get to their destination before and return sooner than the frontier. For the priority, we can set each demand type a different priority according to their level of importance. For example, Resource or System demand can have a higher priority than other demands, and even they come later than a normal demand, the DMS would switch their position in the destination and deliver them sooner.

5.3 Summary

In this chapter, we described the rationale behind having different instances of DMS in GIPSY in addition to detailed information about implementations of each of them using the JINI and JMS technologies. We explained the internal architecture and components of these two instances (i.e., JINI-DMS and JMS-DMS), and we presented different sequence diagrams of these two to show the functionalities and interactions of their components and modules. In the next chapter, we will describe how we evaluate and benchmark these implementations.

Chapter 6

Experimental Investigations

Oh, come with old Khayyam, and leave the Wise,
To talk; one thing is certain, that Life flies;
One thing is certain, and the Rest is Lies;
The Flower that once has blown for ever dies.
Omar Khayyam - Persian Poet (12th century)

In the course of this research, we have perfected our design of the generic Demand Migration Framework which accepts different distributed technologies to apply its rationals and framework criteria to implement Demand Migration Systems. In Chapters 4 and 5, we have explained the design and implementations of our current DMS instances in detail. However, in this chapter, we would like to extend our studies to investigate on the behavior of each of these instances against the elaborated list of quality of services mentioned in Chapter 3.

According to [81], based on the architecture of a system, a middleware contains different components and modules; consequently, it requires detailed observation and studies in measuring the QoS policies. Unfortunately in the GIPSY project, there are many modules that were not ready at the time of our research. Hence, we had

to implement the GIPSY simulator to simulate those unavailable modules and let us observe the behavior of the entire GIPSY system. However, our experimental investigation was designed so that it can be held regard or regardless of the GIPSY simulator.

In this chapter, first we explain the main functionalities of our GIPSY simulator, and the execution environment we did our investigations in. At the end, we explain each of our qualities one by one, and present our experimental results.

6.1 GIPSY Simulator

The GIPSY Simulator is a Java multi-thread distributed computing application that simulates GIPSY in its main functionalities — demand generation, demand migration, and demand computation. The GIPSY Simulator fully integrates our two DMS instances (i.e., JMS-DMS and JINI-DMS) and stubs for instances of Demand Generator and Demand Worker. The latter represent procedural Demand Generator and Demand Worker able to generate and process procedural demands respectively. The Demand Generator stub does not implement any GEER, but a dynamic engine that loads on the fly a predefined order of functional demands. Like real GIPSY, GIPSY Simulator implements hot-plugging, thus allowing the stubs and DMS components to plug in and plug out voluntarily, with no harm to the system's consistency. Moreover, the GIPSY Simulator can work with both of JINI-DMS and JMS-DMS.

Note that both DMS also implement hot-plugging and can be hooked to or unhooked from the system on the fly. In addition, the GIPSY Simulator exposes an open architecture, which allows the stubs to be replaced with other more advanced stubs, or with a complete implementation of the GIPSY Demand Generator or Demand Worker respectively, when ready.

The GIPSY Simulator exposes a Graphical User Interface (GUI) to the users,

who can interact with the system to control the demand generation, migration, and computation process. Over this GUI, users can create execution scenarios by creating profiles that save a sequence of demands to be generated and computed. Using pre-defined profiles allows the same order of demands to be processed multiple times with different configurations of stubs and DMS in different environments (i.e., testing in different kinds of machines or operating systems). The GIPSY Simulator's GUI is attached to the Demand Generator stub, which makes the Demand Generator fully controllable. For example, users can stop or start the generation of demands, dispatching generated demands, or even decide at run-time what demands to be generated. Given the reasoning above, we conclude that the GIPSY Simulator is a testing tool that allows us to test different design and implementation approaches of the DMF in the GIPSY.

As we have started some initial investigations by using the new simulator as our test drive for benchmarking our systems, we found out that using this simulator accomplishes our studies as it gives information about the system behavior in the runtime from the perspective of DG. However, it reduces the performance of our system as it gives additional overhead to the system. These overheads are such as using multi-threading, user interface, etc. Thereupon, we used this simulator only in order to test features such as portability, plugability, and using of co-existing DMS instances for different situations and scenarios. Other features, which test the performance and capacity of our system, we performed them without using the GIPSY simulator.

Although we decided to disregard the simulator in those performance-requiring test cases, we would like to consider this testing tool as an infrastructure that enabled us to study and observe the behavior of our DMS instances as functional components of the system entirely from the very beginning to the end. This tool provided us an

opportunity to analyze our current and projected design approaches.

6.2 Execution Environment

In order to test our system in real life situation, we chose different computers in the Engineering and Computer Science (ENCS) network at Concordia University. They have different configurations, settings, and operating systems, which would give us a very good feedback about the behavior of our systems in certain situations, where computers are all connected to the Internet, and they may face random interaction from/with other computers in their network. Sometimes a technical-support administrator may access one of these computer to do certain procedure or remotely update or install an application or service without any prior notice. We expect this situation in our test case, but unfortunately due to the restricted network policies, we do not have any power to monitor or control these issues.

From the beginning, we considered these situations and unexpected issues, but eventually GIPSY should work in such an environment, so we should foresee these issues and hopefully handle them in a way that none of our GIPSY modules lose their main functionality or performance as soon as they face such a complex environment. In our experimental investigations, we used the computers listed in Figure 1.

6.3 Quality of Service Experiments with JINI and JMS DMS

Hereby, we would like to describe our test cases and results for QoS such as availability, dispatching, latency, throughput, scalability, persistency, flexibility, maintainability, configurability, priority, and portability. At the end of this chapter, we will compare

Table 1: Testing Environment

Machine	CPU	Speed	RAM
Hafez(LAN)	PC Pentium IV	3.01 GHz	512 MB
Rumi (LAN)	Pentium T2400	1.83 GHz	2.00 GB
Namibia.encs.concordia	2 Core CPU 6300	1.86 GHz	2.00 GB
Mozambique.encs.concordia	2 Core CPU 6300	1.86 GHz	2.00 GB
Zambia.encs.concordia	2 Core CPU 6300	1.86 GHz	2.00 GB
Lesotho.encs.concordia	2 Core CPU 6300	1.86 GHz	2.00 GB
Zimbabwe.encs.concordia	2 Core CPU 6300	1.86 GHz	2.00 GB
Malawi.encs.concordia	2 Core CPU 6300	1.86 GHz	2.00 GB
Tanzania.encs.concordia	2 Core CPU 6300	1.86 GHz	2.00 GB

the technologies we used in our implementations.

6.3.1 Availability

We check the system's availability in the sense which one could start up the system faster to function properly. For example if a system takes seconds to be ready for accepting incoming request from other nodes, it would be more desirable than the one which takes minutes.

This aspect is related to their distributed middleware system (i.e., JINI and JMS) rather than our own implementation in DMF. As mentioned before, each of theses two systems should run certain services in order to be active. The longer it takes to run those services, the lesser available is that instance of DMS.

We benchmarked our system in an environment where DG constantly generates demands in a fixed interval, but on the other side, the DMS node shuts down and automatically re-initialize its required services to distribute demands. The purpose of this test was to see which of these instances reaches the fully functioning state faster, or generally speaking to see which one requires lesser time to start its services.

Table 2 shows our result for the starting up of the DMS instances in seconds. For

Table 2: Startup time in JINI-DMS and JMS-DMS

Attempts	JMS-DMS (sec.)	JINI-DMS (sec.)
1	14.66	180
2	14.23	180
3	14.09	60
4	14.21	75
5	14.70	300
6	14.12	180
7	14.50	240
8	14.59	240
9	14.48	60
10	14.66	240
11	14.06	70
12	14.36	300
13	14.01	180
14	14.08	240
15	14.47	300
16	14.14	65
17	14.01	450
18	14.01	70
19	14.03	300
20	14.88	97

JINI-DMS, JavaSpace, JINI lookup service, and `RMIRegistrary` should be started, whereas in JMS-DMS, only the application server(i.e., Jboss) should be running. As can be seen in Table 2, we calculated the runtime of our DMS instances many times. The clock was set to calculate from the request of initializing services to the final point where DMS is ready to respond to any incoming demands.

In JMS-DMS, all requests has been responded, whereas in JINI-DMS 29 of our requested have been refused as JINI-DMS was not fully initialized to perform completely. As can be seen in Table 4, we continued our test in JINI-DMS to have at least 20 results like the other instance.

Figure 30 illustrates that JMS-DMS starts its services way faster, on average 14.28 seconds, whereas in JINI-DMS takes a couple of minutes (average of 191 seconds) in

Table 3: Minimum, Maximum, and Average Startup time of both DMS instances

DMS Instance	Minimum Time (sec.)	Maximum Time (sec.)	Average Time (sec.)
JMS-DMS	14.01	14.70	14.28
JINI-DMS	60.00	450.00	196.32

Table 4: Availability Rate in JMS-DMS and JINI-DMS

DMS Instance	Total Attempts	Failure	Successful	Availability Rate	Average Up Time(s)
JMS-DMS	49	29	20	40.82	191.00
JINI-DMS	20	0	20	100.00	14.28

most of the cases. We also summarized the minimum, maximum, and average startup time (in seconds) of each of them in Table 3.

By verifying the data in Table 2, we noticed that these two instances have different rates of availability, which we demonstrated in Table 4.

This poor performance of JINI-DMS is because of the initializing of registry services to start a working JINI/JavaSpace system and also the good feature of lookup service of JINI, which plays against this QoS in here. Definitely JINI lookup service would improve the scalability of the JINI-DMS as it can discover any new node or TA across its network. However, this can be eventually a time-consuming task, which in some cases may not be necessary.

6.3.2 Dispatching

In order to transfer demands from one node to another, DG should establish certain services to convey demands toward the DMS. This time consists of receiving the request for a certain demand, wrapping them with additional information (e.g., its current state, timestamps, and identifications), establishing a successful connection with the DMS, and finally dispatching them into a carrying channel. As represented in Figure 31, we consider the combination of these steps as the dispatching time. Here,

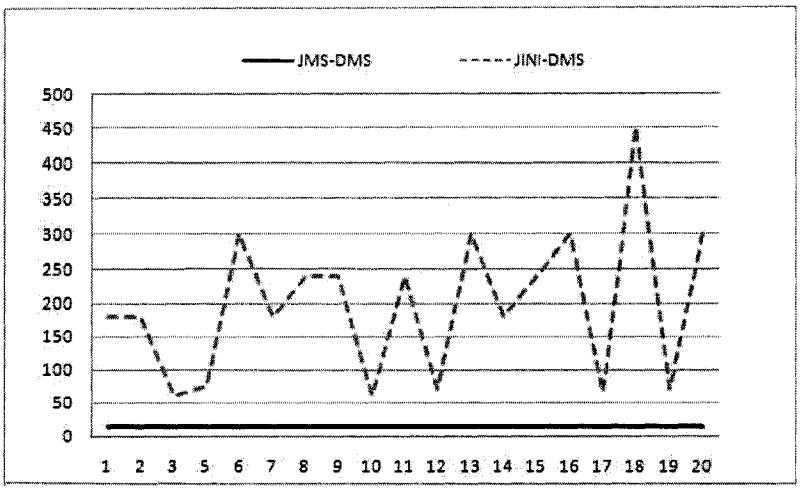

Figure 30: Startup time for JMS-DMS and JINI-DMS

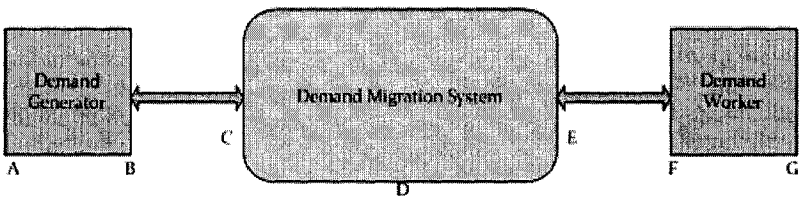

Figure 31: Time sequences in GIPSY DMS

we would like to study the behavior of our system in terms of dispatching demands from small numbers of demands to an order of thousands of them in each request.

We represent the dispatching time as the difference between T_B and T_A, where T_A represent the exact time of initiating a new demand, and T_B is after the time the connection is properly established and the demand is at the point of leaving the DG.

$$T_{Dispatching} = T_B - T_A \qquad (1)$$

Implementation-wise, our wrapping mechanism follows the same approach in both DMS instances, but the main difference is the time each requires to establish a successful connection as the means of their communication. The faster they establish this means, the quicker they can effectively send out demands, which eventually leads to a better dispatching time.

Consequently, we took this into consideration, and tested the dispatching time in our DMS instances. As shown in Table 5, we generated batches of demands containing small numbers of demands up to the heavy load of 45000 of them in a single batch.

In Table 5, the total dispatching time, in milliseconds, is the total time the whole batch requires to dispatch its requests within an open connection. Dispatching/Demand represents the average of that total dispatching time per a demand in each batch.

As illustrated in Figure 32, comparing to JMS-DMS, JINI-DMS takes longer to dispatch a demand at the very beginning. For the first demand, it took 94 milliseconds as JMS-DMS took nearly half of it (i.e., 47 milliseconds). This is due to different services JINI should run in order to establish a connection, so it eventually becomes more expensive in terms of consuming more resources in very small numbers of demands.

Due to the open connection in each procedure of sending out demands, in larger numbers of demands in each request, the average dispatching time would be higher, but eventually the average dispatching time per demands would be lower as this time should be divided by total numbers of demands in each batch.

As can be observed, at the beginning of this experiment, with small numbers of demands, the average dispatching time per demand is at the highest point, whereas from 30 demands to more, we see considerable reduce over that average time. Interestingly enough, as we reach 10000 demands, JINI-DMS increases the dispatching time once again very rapidly, whereas in JMS-DMS the average time would be decreased. This shows that those time-consuming tasks for establishing a connection in JINI-DMS become quite expensive. Thus, JMS-DMS places in a better position in terms of dispatching time in both small and larger number of demands.

Table 5: Dispatching time in JINI-DMS and JMS-DMS

	JMS-DMS (ms)		JINI-DMS (ms)	
Numbers	Total Dispatching Time	Dispatching/Demand	Total Dispatching Time	Dispatching/Demand
1	47.00	47.00	94.00	94.00
5	62.33	12.47	172.00	34.4.00
10	83.33	8.33	260.33	26.03
15	99.00	6.60	286.33	19.09
20	104.33	5.22	333.33	16.67
25	130.33	5.21	369.67	14.79
30	140.33	4.68	427.00	14.23
35	166.33	4.75	484.00	13.83
40	166.67	4.17	520.67	13.02
45	182.33	4.05	562.33	12.5
50	187.33	3.75	661.00	13.22
60	217.67	3.63	677.67	11.29
70	234.00	3.34	760.33	10.86
80	250.00	3.13	885.00	11.06
90	291.33	3.24	989.33	10.99
100	297.00	2.97	1114.33	11.14
150	390.00	2.60	1572.33	10.48
200	515.33	2.58	2126.00	10.63
250	614.67	2.46	2671.33	10.69
300	671.67	2.24	2724.33	9.08
350	942.67	2.69	3113.67	8.9
400	1036.67	2.59	3446.33	8.62
450	1031.00	2.29	3640.33	8.09
500	1172.00	2.34	5358.67	10.72
550	1729.67	3.14	5790.67	10.53
600	1542.00	2.57	7534.33	12.56
650	1655.67	2.55	9060.33	13.94
700	1906.00	2.72	7691.00	10.99
750	1586.67	2.12	9226.67	12.30
800	1677.00	2.10	7498.00	9.37
850	2458.33	2.89	10114.67	11.90
900	1880.33	2.09	8484.33	9.43
950	1942.33	2.04	7875.00	8.29
1000	2031.00	2.03	11072.67	11.07
2000	4260.33	2.13	18901.67	9.45
5000	11880.67	2.38	48943.00	9.79
10000	23019.67	2.30	136648.00	13.66
15000	33168.67	2.21	206587.33	13.77
20000	42922.67	2.15	290994.00	14.55
25000	54884.33	2.20	386496.00	15.46
30000	64038.33	2.13	469539.67	15.65
35000	74088.67	2.12	551577.33	15.76
40000	82496.33	2.06	598736.00	14.97
45000	92336.67	2.05	693775.67	15.42

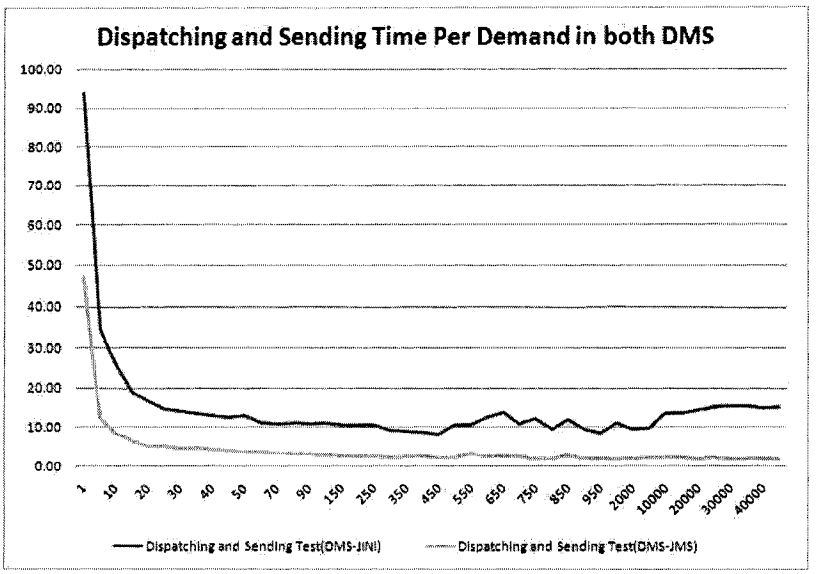

Figure 32: Comparison of dispatching time between JMS-DMS and JINI-DMS

6.3.3 Latency

Previously in section 3.3.2, we defined the term latency in quality of services in distributed systems. However, in this section, we would like to explain our approaches for the measurement of this aspect in GIPSY.

In our GIPSY environment, we started the process of demand migration from one node to another node in our network. In order to measure the time elapsed from the point a DG sends out a demand to the point a DW receives that request, we measured different timestamps in our implementations.

As shown in Figure 31, and represented in formula (2), latency is defined as the time it requires to traverse a demand from DG to the DMS ($T_C - T_B$) plus the processing time in the DMS ($T_{Processing}$), in addition to the time it requires to migrate a demand from DMS to a DW ($T_F - T_E$). Processing time (($T_{Processing}$)) consists of the time DMS requires for prioritizing a demand, indexing the order of incoming demands from different DGs in its communication channels, and storing them in its persistent storage.

$$T_{Latency} = (T_C - T_B) + T_{Processing} + (T_F - T_E) \qquad (2)$$

As Jboss application server and JavaSpace are both bottlenecks of our DMS, by adding more demands and DW, definitely we can observe different behaviors from them. It should follow a linear curve at the beginning, but might not follow in such an expected way as we receive high loads of demands as our resources such as hardware, virtual machine, or operating system are limited. Consequently, in order to study the behavior of our systems in different situations, we started our demand migration with a small number of demands, and increased that number up to 45000 in each of our batches.

Faison [45] outlined in some of the main difficulties everyone can encounter in the measurement of the latency as it is bound to two key parameters of network traffic and bandwidth. Therefore, in GIPSY, we would like to see the behavior of our DMS instances in a very complex but close to real-life situation network. As explained in section 6.2, functionality and behavior of our DMS in such an environment is very important in our experimental research.

Moreover, measuring latency helps us understand the availability of our DMS instances in facing high loads of demands, and how well they perform when numbers of demands in their communication channels are high. Having said that, high number of demands is not an independent factor, as a very high number in one system can be a small one in another one, but this would not stop us from doing our experimental investigations, as we are comparing these two systems within their default configurations without adding any additional database, clustering infrastructure, or network speed booster. Thus, considering these situations, our investigation is still valid and helpful for our further steps toward improving our systems to accept much more complex scenarios and situations.

As said before, we have to measure the time a demand may require to migrate from

a DG to a DW considering numbers of working load in its communication channel. As shown in Figure 31, we should measure different timestamps in each route of demand migration. However, the ways we measure these are different in our DMS instances.

In JMS-DMS, JMS provides us the timestamps variable in message headers, so we can simply set it in each point during this procedure and read in the next step, whereas in JINI-DMS, we should set the timestamps as we are wrapping a demand with additional information on top it. One of those additional information can be the timestamps the can be set in DG and be read along the way.

Another complexity of measuring the latency, is the time synchronizations, which would make this test more complicated in web-based systems. However, in our situation, we are doing this test in our internal Local Area Network (LAN) where all the clocks are automatically synchronized by the main server.

Chen [30] followed an empirical approach to test the network latency in his infrastructure, but he limited himself to the size of messages from (50 B. to 4 K.B), whereas in our case, we extend his approach in both terms of small-sized and large-sized demands, which we would mention in more detail in 6.3.5.

As shown in (3) we did each of our test three times, and finally by dividing the average of these three measurements by the number of demands (i.e., N) in each batch of requests, we measure the latency time per demand (i.e., Avg.Latency/demand) in our DMS. At the end of our measurements, we evaluated the minimum, maximum, and average latency time of our DMS instances (see Figure 7).

$$T_{Avg.Latency/Demand} = Average(T_{Latency_1} + T_{Latency_2} + T_{Latency_3})/N \qquad (3)$$

Hence, for each DMS instance, in Table 6 the first column is the average latency of our three measurements, and the second column in the average latency time per a demand $T_{Avg.Latency/Demand}$. All our time calculations are in milliseconds in this test case.

Table 6: Latency time in JINI-DMS and JMS-DMS

	JMS-DMS		JINI-DMS	
Numbers	Avg.Latency (ms)	Avg.Latency/Demand (ms)	Avg.Latency (ms)	Avg.Latency/Demand (ms)
1	67.67	67.67	85.23	85.23
5	320.67	64.13	394.40	78.88
10	594.33	59.43	786.99	78.70
15	885.33	59.02	1113.50	74.23
20	1148.33	57.42	1412.38	70.62
25	1422.33	56.89	1749.38	69.98
30	1624.00	54.13	2054.42	68.48
35	2039.00	58.26	2475.85	70.74
40	2026.33	50.66	2792.26	69.81
45	2464.67	54.77	3031.39	67.36
50	2511.33	50.23	3588.78	71.78
60	3189.00	53.15	3922.28	65.37
70	3587.00	51.24	4631.79	66.17
80	4214.67	52.68	5183.79	64.80
90	4763.00	52.92	6008.20	66.76
100	5189.67	51.90	6499.98	65.00
150	8235.00	54.90	9908.55	66.06
200	11519.67	57.60	14168.50	70.84
250	14858.33	59.43	18994.85	75.98
300	17950.33	59.83	22077.82	73.59
350	20335.00	58.10	25998.82	74.28
400	23292.00	58.23	30247.75	75.62
450	27873.67	61.94	33282.93	71.96
500	32217.00	64.43	37124.96	74.25
550	34034.67	61.88	42102.82	76.55
600	35751.33	59.59	43971.97	73.29
650	38592.00	59.37	49860.58	76.71
700	42548.33	60.78	52331.87	74.76
750	46090.67	61.45	56688.73	75.58
800	49047.67	61.31	60325.66	75.41
850	54733.67	64.39	64319.10	75.67
900	55113.67	61.24	68196.47	75.77
950	57530.67	60.56	70759.24	74.48
1000	61121.67	61.12	75175.95	75.18
2000	131351.33	65.68	145784.66	72.89
5000	401830.00	80.37	357061.34	71.41
10000	766147.00	76.61	729911.66	72.99
15000	1139673.67	75.98	1082010.71	72.13
20000	1549004.00	77.45	1473097.43	73.65
25000	1976074.00	79.04	1828979.50	73.16
30000	2382987.67	79.43	2200960.75	73.37
35000	2666399.67	76.18	2737534.28	78.22
40000	3187948.33	79.70	3103453.09	77.59
45000	3726571.00	82.81	3785708.65	84.13

According to [76] the worst-case execution times should not be taken at face value, as the actual value is largely dependent on the operating system scheduling and network latencies. They showed that worst-case behavior should be attributed to the operating system and network, thus cannot be used for comparing different middleware services. Therefore, we removed the worst-case situations out of our measurements for both of our DMS instances.

We show the result of this study in Figure 33, where the vertical axis is the latency time in milliseconds and the horizontal one is the number of demands per batch. We started with very small numbers to observe the behavior of the system at frequent situations, and step by step, increased the number of demands up to 45000 to observe the performance of the system in facing of transferring larger numbers of demands.

As shown in Figure 33, regardless of the fact of having better dispatching time in JMS-DMS, comparing to JINI-DMS, JMS-DMS performs better at the beginning of our experiment. Basically at the beginning, JMS-DMS delivers each demand about 20 milliseconds faster than the JINI-DMS. They both keep the same linear behavior within the somehow fixed difference between their latency time up to the batch of 2000 demands. JMS-DMS, contrary to JINI-DMS, suddenly takes longer time to deliver that batch, whereas in JINI-DMS it continues as expected linear behavior. Later on they both increase their average latency time per demand relatively. Figure 33 has some interesting points that we would like to emphasis more:

- The difference of 20 milliseconds per demands up to the batch of 2000 puts JMS-DMS in a better position in terms of performance, as it can eventually perform faster about 40000 milliseconds for each batch of 2000 demands. As a result, JMS-DMS would be a better solution for batch of 2000 demands or less.

- Since JMS-DMS uses the default embedded HSQLDB as its persistent storage, it is not suitable for under-pressure situations, whereas in JINI-DMS, it has the

Table 7: Minimum, Maximum, and Average Latency time of both DMS instances

DMS Instance	Minimum Time (ms)	Maximum Time (ms)	Average Time (ms)
JMS-DMS	50.23	82.81	62.59
JINI-DMS	64.80	85.23	74.23

benefit of using a stronger embedded database of JavaSpace. In our situation, we can see that JMS-DMS is under pressure of workload as it reaches 2000 demands in a single batch. Even though it may be rare to have situations of 2000 demands in every single batch in our daily basis, but we would like to challenge our systems in different situations where the understudy scenario escalates up to some extends.

- After 2000 demands, both of them increase their latency time, but the interesting part would be where HSQLDB performs considerably unexpected than the other system. Thus, we would like to regard this exactly as the paging mechanism in operating systems, where at the point of exceeding their page size, their performance would be declined to accept and buffer incoming demands.

- In this test, we found out that this test is basically challenging their databases, and both declines and inclines in this diagram would mostly reflect the performance of their embedded databases. Hence, we stop our test case at this point, as this is not related to performance of the entire system, and the database can be replaced at anytime.

Just to summarize both diagrams and result tables for this test case, we provide a simpler table (see Table 7) that shows that generally these two systems somehow perform similarly.

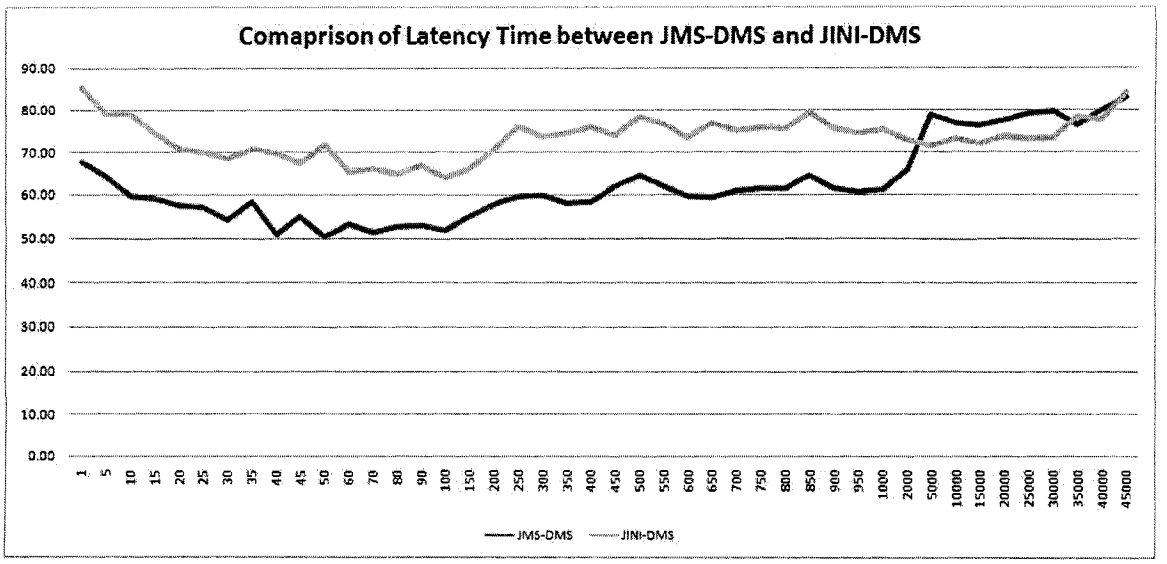

Figure 33: Comparison of the latency time between JINI-DMS and JMS-DMS

6.3.4 Latency + Dispatching

As part of our research, we took another step to investigate more on latency and dispatching in our DMS but this time with a different approach. This time, we chose JMS-DMS as the newer system to be our playground for this study, and we provided an infrastructure to explore both of these services. In previous cases, we have located the DMS process in the same computer as the DW, but in this section, as represented in Figure 34, we would like to change the place of DMS process from one node to another to observe the impact it may have on the behavior of the JMS-DMS. We located DMS in scenario A in the node of DW, and in scenario B, in DG side, and finally in the last scenario, we located each in a separate node.

For all of these three scenarios, we studied the latency and dispatching time by sending different demands in every single request starting from 10 to 500. Table 8 shows the average latency time per demand for each of our scenarios, which is the result of repeating our tests cases three times and using their average per numbers of demands in this table, and subsequently, in Figure 35.

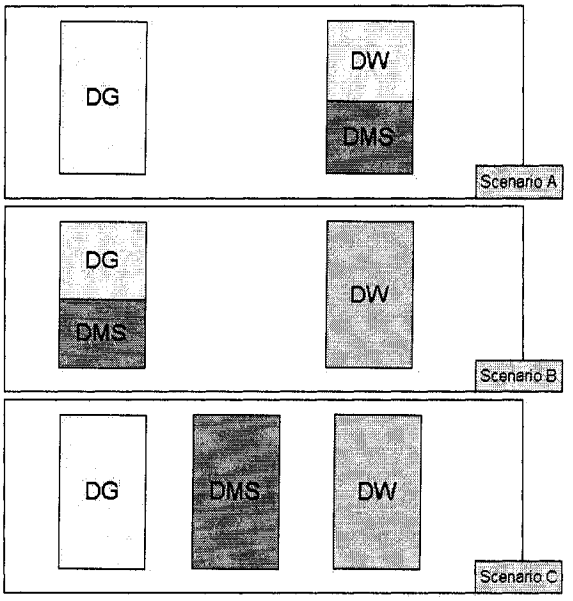

Figure 34: Testing scenarios of DMS distribution for latency measurements

Table 8: Average Latency Time per Demand in Different Scenarios

Numbers	A (ms)	B (ms)	C (ms)
10	339	155	394
50	338	76	477
100	282	75	450
150	270	63	448
200	253	65	460
250	257	62	444
300	260	64	452
350	268	64	448
400	269	62	447
450	282	62	462
500	307	63	461

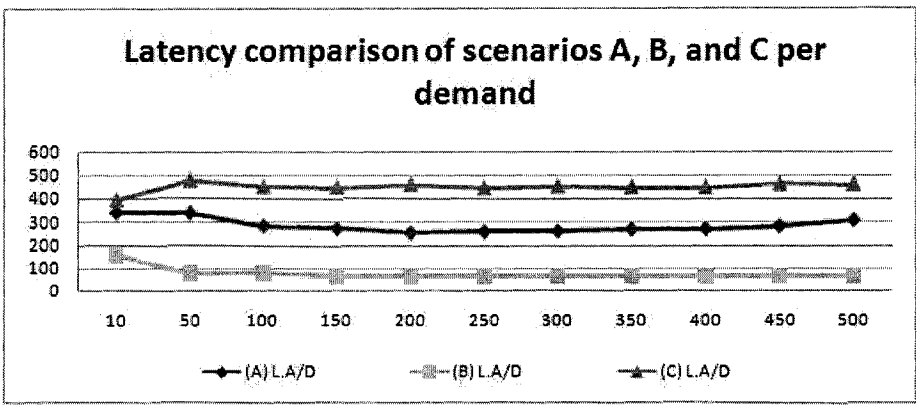

Figure 35: Latency comparison in different scenarios of DMS distributions

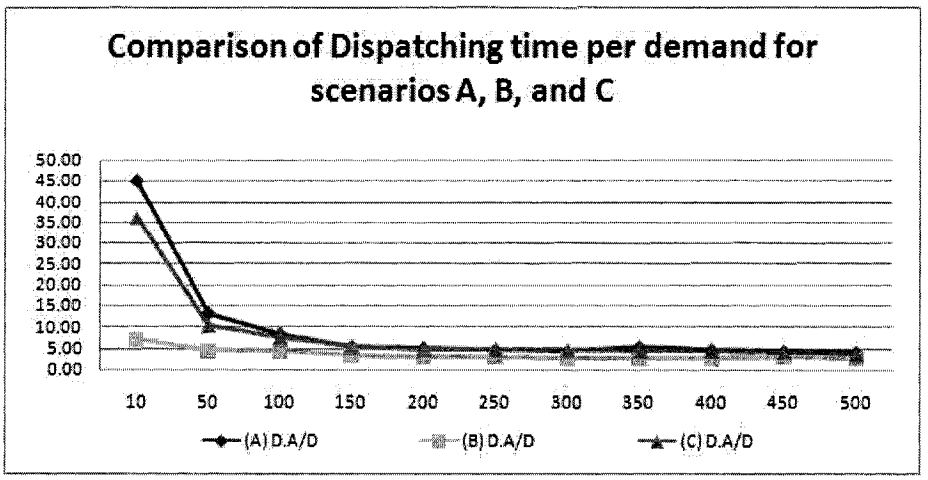

Figure 36: Dispatching-time comparison in different scenarios of DMS distributions

As also represented in Figure 35, for latency, the B scenario takes the least amount of time to transfer demands as DMS resides in the same node as DG. Locating the DMS on DG side, helped the `Demand Generator` to send all its requests toward the DMS faster comparing to other scenarios. Between other scenarios, A takes less amount of time to transfer demands, as there is no network latency between DMS and `Demand Worker`.

We applied the same approach for the average dispatching time in all these scenarios. We did the test case up to 500 demands in a batch for three times, and used the average of these three for Table 9 and eventually Figure 36.

Table 9: Average Dispatching Time per Demand in Different Scenarios

Numbers	A (ms)	B (ms)	C (ms)
10	45.02	7.05	36.10
50	12.98	4.37	10.28
100	8.18	4.15	7.58
150	5.22	3.22	5.42
200	4.89	2.81	4.69
250	4.42	2.72	4.75
300	4.06	2.63	4.56
350	5.09	2.64	4.40
400	4.49	2.62	4.39
450	4.12	2.81	4.02
500	3.85	2.50	3.58

As also represented in Figure 36, the best case for dispatching time is again the B scenario, as it connects way faster to the DMS as expected. However, both A and C perform somehow similarly as both interact with DMS remotely for their initializations.

In real-life scenarios, we can have any of these distributions in DMS, so the distribution of DMS is directly bound to the requirement of the system or the execution environment. The main purpose of this test was to study the behavior of the DMS in such environments.

6.3.5 Throughput

Having defined throughput in section 3.3.2, we would like to explain in detail how we studied this aspect of QoS in the course of this research.

By studying both instances of DMS, we found out that they both suffer from certain limitation and obstacles. Their behavior and performance change as we overtake some experimental investigations especially upon reaching some specific points where either of those performs unexpectedly. According to [30], by keeping track of

those limitation points and trying to control the system not letting to exceed that, we can run the test ideally forever, but realistically without having any of our queues or buffers oversize. Chen et al. [30] calls this point as the Maximum Sustainable Throughput (MST) metric of the system.

MST is to quantify the maximum throughput of a messaging server/provider, so technically speaking, by controlling the system not to reach those points, or furthermore, by injecting some flow-control layer which distributes the load of the system without letting a specific parts exceeding its MST, we can have our system performing without having any uncontrollable load or pressure. However, implementing such a layer was not outlined in our current research agenda, and we would like to implement the injection of flow-control layer in our future work.

So far, we tested our systems against variable size and numbers of demands in each batch, but for this part, we would change the size of demand in each migration to see how our DMS would behave in facing different sizes of demands. We continue our test to see how it escalates in situations where each demands carries data, code, and sometimes objects. In order to simulate such a situation, we instantiate demands which in their computation process take a screenshot of the remote machine's (i.e., DW) desktop. A remarkable feature of these demands is their size, i.e., we can test the capacity of the DMS throughput against the size of demands. We can increase the size of the demand incrementally by increasing numbers of screenshot in one demand.

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 returned to the DG, which has issued this demand. The DG request numbers of screenshot, and upon reception of that demand, the DW start capturing snapshots, and returns those graphical results to the DG in one batch.

We represent the result of this study in both Table 10 and Table 11. We have

Table 10: Throughput Results for JINI-DMS

Demand Size (Kb)	Average Time (ms) per 1 Kb
79.40	43.6398
153.20	33.13969
230.10	35.03259
307.60	37.31144
498.60	37.15804
581.70	35.39453
704.80	34.39983
792.00	32.14141
881.00	33.69202
1321.50	33.69202
1768.00	36.91968
2217.50	44.32514
2718.00	40.40029
3696.00	38.45617
4630.00	44.81814
5562.0	61.97681

done each of these test cases three times, and put the average of these three as the time in both given tables.

Corresponding to Figure 37, the numbers on the X-axis represent the average time in milliseconds needed to migrate 1 Kb of a screenshot demand, where the demands are constantly growing in size, and the Y-axis represents the demand size in Kb.

In this test, we used both DMS instances to migrate large-sized demands (in both directions) with a size increasing up to 6000 Kb (approximately 6Mb). For JINI-DMS the migration was successful for all the demands sized up to 5562 Kb (approximately 5Mb), and somewhere around 6Mb the system raised a memory exception `java.lang.OutOfMemoryError`, which is thrown when the Java Virtual Machine (JVM) cannot allocate an object due to lack of its heap size memory. For JMS-DMS the migration was successful for the demands sized up to 2310 Kb (approximately 2Mb), and somewhere around 2.5Mb the system raised the same memory exception.

Table 11: Throughput Results for JMS-DMS

Demand Size (Kb)	Time (ms) per 1 Kb
79.00	36.10127
154.00	36.52381
231.00	31.68831
324.00	28.15741
385.00	32.41039
462.00	34.11905
539.00	39.66790
616.00	39.16234
693.00	34.81097
770.00	40.25974
1155.00	37.66234
1540.00	38.28896
1925.00	56.93818
2310.00	59.03290

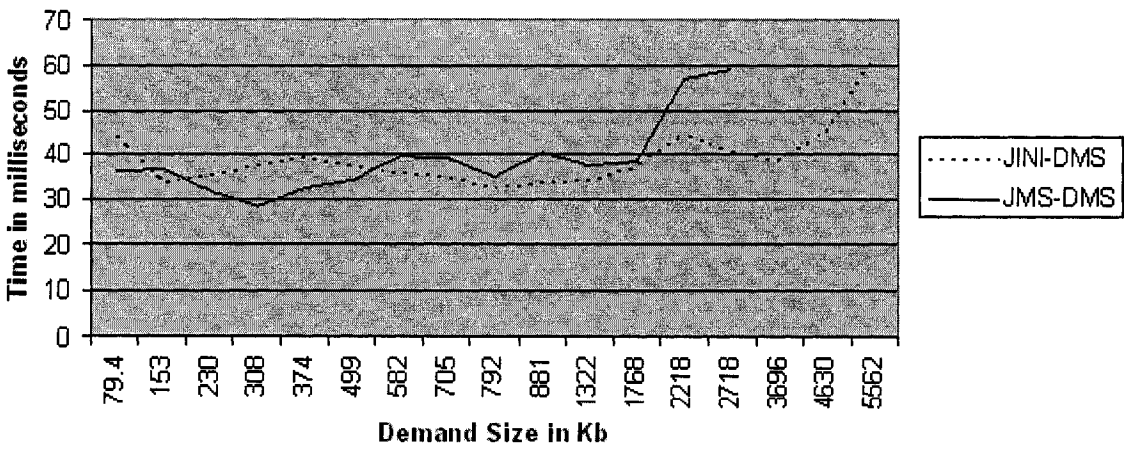

Figure 37: Comparison of throughput between JMS-DMS and JINI-DMS

We initially overcame this problem by simply increasing the size of JVM heap memory. Therefore the limit seems to be the limitation of the physical memory on the machine under consideration. Therefore, both instances show high capacity for migrating big-sized demands, but both require additional maintenance intervention to increase their capacities.

The curves shown in Figure 37 depict the time-size correlation in the migration of big-sized demands. For JINI-DMS, the curve shows an average time around 40 ms up to 4500 Kb and after that there is a steady increase. This makes JINI-DMS reliable up to 4500 Kb. However, for JMS-DMS the curve shows an average time of 35 ms up to 1800 Kb, and then it goes straight up to 60 ms.

In addition to the extension of the heap-size memory, we can improve the throughput of both DMS instances by modifying their persistent storage as they play an import role in storing and retrieving demands especially when it comes to larger ones. As can be observed in Figure 37, JINI-DMS takes advantage of using more stable persistent database for storing its demand comparing to JMS-DMS. Therefore, JINI-DMS stays at the more productive, and also much reliable position with messages sized over 2200 Kb, whereas JMS-DMS is better performing with messages below that size.

6.3.6 Persistency

Previously, we defined the term persistency in 3.3.2, and here, we explain how we are going to measure this factor in our QoS. As mentioned previously, we need to store our demands in some sort of a persistent storage in order not to lose our demands in case of any failure. This would make this aspect of QoS very important especially to see if our instances perform well in this term.

In order to find out the performance of this aspect, we had to simulate the effect

of various failures. These failures can be shutting down the main server computer, logging off from the operating system of the server computer, or even closing off the DMS application in the middle of demand-migration procedure. Even though all of them raise some unexpected error, each of these has different effect on the execution of the middleware.

Non-persistent messages will eventually fill up the memory made available for queues, and persistent messages will run out of allocated storage space — and both situations pose risks to the stability of the message server and the entire distributed system. As opposed to this problem, in the future, we need to have a flow controller to increase the load balancing of the DMS, which was not part of this thesis.

Here, we are going to explain different scenarios that we would like to test our system against them to find out if our systems are persistent or not. At the end, we discuss the overhead the persistency may bring into our system.

We performed these test cases by generating numbers of demands (i.e., 100 demands) in one batch, and triggering one of these failures, and observing the behavior of the DMS in facing these problems. We used some remote machines (four DWs and four DGs), and we located DMS in one of them. After doing this test cases in each of these three types of failures, we found out that both of our DMS support persistency, as we measured both numbers of generated and received demands in both DMS instances.

Shutting down the server

By shutting down the server node, we are simulating a non-protected network without any kind of additional power supply for the time of power shortage. It does not let the system save any work, so it has the highest risk of data lost comparing to other types of failures.

In order to test our system against this, we start our demand migration and in the middle of its work, we suddenly turn off the computer by unplugging it from the power. There are many pending and computed demands waiting to be routed to their destination. After a while, we re-start the failed node and re-establish the connection to the DMS again, and we see that the demand migration continues very soon.

Logging off the server

Logging off the operating system of the DMS machine is a moderate disaster comparing to shutting down the server, as it gives more time to the server and application to save some pending messages. It is not as harsh as the previous one, but still provides us with helpful information about the persistency of these two versions. For this test, we log off the server operating system (e.g., Windows) in the middle of the demand-migration process.

Closing off the middleware application

If a message is on its way to the DMS and suddenly the DMS is down because of any kind of failure or manual shut down, it does not have any place to store. It is very interesting to see the behavior of pending messages, and find out where and how DMS stores them, and how we can retrieve them afterward. Even though, it takes a while to continue their routine, it eventually transfers all pending demands.

We noticed that at the time of the occurrence of any type of failures, DMS stores all demands permanently, and as soon it restarts to perform again, it continues the procedure by sending all demands stored in its physical storage. On the other side, either of DW and DG have valid connection to the DMS and as soon as the connection is not valid anymore or demands face failed DMS on their ways, they acknowledge those execution nodes to stop sending any more demands toward the failed DMS.

Upon re-establishment of connections between GIPSY execution nodes and DMS, they continue their routine by sending pending demands.

Message Size versus Persistency

Demands which transfer from one node to another in a persistent mode, carry over information (e.g., persistent-storage hosting location etc.) in addition to their functional content.

We did some investigation to dissect if such a communication mode affects the size of demands or not. By verifying the documentations of HSQLDB, we could not find any standard approach to measure the size of demands. As both of them stored messages in their local drive, we measured the size of that specific directory before and after the demand migration.

We considered the demand size as the difference of this size divided by the number of generated demands. As we showed the results in Table 12, we generated 300, 500, 600, 700, 900, and finally 2000 demands to measure the size of each demand in each of these migrations. We could see that persistent demands are mostly larger than non-persistent demands because of these additional information.

Interestingly enough, as shown in Figure 38 and Table 12, we noticed that the size of each persistent demand increases as the number of demands in a batch increases. However, it should be taken into the consideration that this is not the actual size of each demand, but just the average size of each demand stored in the system. Thus a demand can be only 7 Kb, but when we have a batch of 500 persistent ones, including those additional information for each demand and more importantly for the system to store all of them, it can be 24 Kb.

In section Figure 38, the vertical axis is the average of total size per demand and the horizontal axis is the number of demands in each batch.

Table 12: Persistency versus Demand Size

Number of Demands	Non-Persistent(K.B)	Persistent(K.B)
300	7.10	18.33
300	7.09	17.67
300	7.09	19.00
Average	7.09	18.33
500	7.14	24.00
500	7.07	26.00
500	7.06	22.00
Average	7.09	24.00
600	7.06	26.67
600	7.23	25.00
600	7.11	24.17
Average	7.13	25.71
900	7.33	26.11
900	7.12	25.00
900	7.11	26.67
Average	7.19	25.93
2000	7.12	26.00
2000	7.13	27.00
2000	7.14	32.50
Average	7.13	28.50

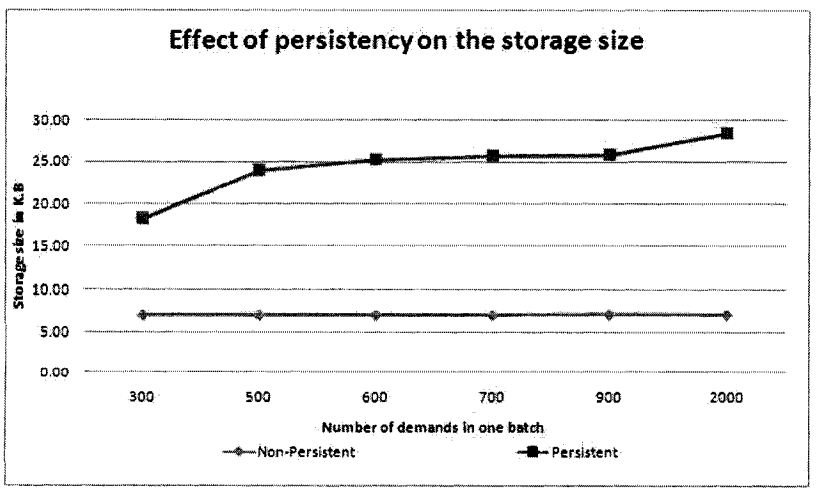

Figure 38: Effect of persistency on the size of demands

6.3.7 Flexibility and Maintainability

In 3.3.2, we explained why we need flexibility and maintainability in our infrastructure, and here we provide more detail about benchmarking them in our two DMS instances. Flexibility involves the level of configurability and ease of use in application level, and maintainability how we can maintain and comprehend our implementations. In the following sections, we discuss these various aspects.

Configurability

DMS needs to be dynamic and configurable. In order to fit in different cases, modification in the configuration of the DMS is somehow necessary. Configurability itself can be studies in two levels of lower and higher.

Lower-Level Configuration This refers to the ability of modification and configuration in their implementations in terms of modification and refactoring. The level we can modify or refactor classes and libraries of these two instances. As both DMS instances have been developed in our lab, we have access to all related classes of our DMS. However they both used JMS and JINI which are not done in our lab. These are open-source APIs that we use in our implementation, and the only way to modify them is to build a modified versions of those libraries. So implementations-wise, both of our DMS instances are open to any sort of modification or low-level configuration.

Higher-Level Configuration Both instances use external application to handle their `Demand Store` mechanism. Therefore, we need to examine JavaSpace in JINI-DMS and JBoss Application Server and its embedded database HSQLDB in JMS-DMS. During our development, we have noticed that we have access to the lowest level of configuration in Jboss Application Server, and we can easily add, modify, or delete any destination or internal configuration inside it. We could also switch from

one version to another as the configuration is similar in all versions of Jboss. We could change everything through their simple GUI or , as we did, by writing codes to apply our changes automatically. For JINI-DMS, we can have this level of configurability by modifying certain classes inside JavaSpace implementation.

Ease of Use

By the batched setup file that we have prepared for each of these instance, their installations and executions are very simple and easy in any environment. However in order to modify them, a user needs to have some basic knowledge of JINI or JMS. As we have designed both of these instances by applying standard design patterns such as facade and adapter, it would be easier to understand and modify if a user has such knowledge.

Maintenance at the code Level

As mentioned in the ease of use, we have included different design patterns in our design. Especially in the implementation of some important classes such as TA, we dealt with them as abstract as possible, so other classes and sub-modules of the system can interact with them just by calling their method names and passing some required arguments without knowing any internal functionalities of them. Therefore, we can modify and refactor our code without changing anything inside the TA implementations of our DMS instances. Consequently this level of abstraction makes our implementation maintainable without interfering or changing any other classes.

6.3.8 Priority

As already mentioned, priority plays an important role in improving our performance. In JMS-DMS, priority mechanism is embedded inside the JMS technology as part of

a header message, so we can set the priority number between 1 and 9 (1 as the lowest and 9 as the highest). Therefore, it would be very easy to provide this aspect in our DMS. In JINI-DMS it is not included in the JINI technology, so we need to implement this feature. In order to prioritize a demand in JINI-DMS, we should tag each message with its level of priority while `DispatcherEntry` is wrapping them on the DG side. Consequently, in situations of having various DG, we need to open, verify the buffers or communication channels on the way of demands from DG to their shared database (i.e., JavaSpace). This browsing mechanism is not only complicated but also very time consuming. Therefore, we would consider the priority mechanism in JMS-DMS easier and faster to configure, use, and expand.

As mentioned before, different kinds of demands have different level or priority. For example a resource demand should be delivered to the destination faster as the provided resource through this demand would increase the performance of the entire system, whereas in a procedural demand, it is mostly not related to the entire system.

6.3.9 Portability and Plugability

As mentioned before, these two elements are very important in our infrastructure. As our application may be used in different operating systems, we should examine if we can deal with these systems properly. As they both require certain libraries, services, and JDK (Java Development Kit) in order to compile and function. We have to examine if we can use these two instances in a plug-and-play manner, so we do not need to worry about any additional installation in our network, and we can start using our DMS modules immediately.

As can seen in our execution environment as presented in Section 6.2, we tested both of our DMS instances in a network which is very close to the real-life scenario. It consists of different operating systems (e.g., Windows and LINUX), storage space,

speed, etc. By including all required files and batch files, we can work with these two instances as a plug-and-play installation in any sort of environment. The only thing that needs to be set is the basic Java configuration.

In order to test the plugability of our systems, we shut down our DMS instances, or any of our execution nodes such as DG or DW to see if the entire system continues performing or not. We also shut down a TA and again restart from another remote node to see if it functions or not. Both of our systems functions properly in all of these situations and find the newly added node.

6.4 Comparison of JINI and JMS Technologies

In previous sections, we compared our two instances of DMS (i.e., JMS-DMS vs. JINI-DMS), but here, we would like to compare these two in terms of their technologies regardless of the DMS. Previously, we outlined their positive points and features in Section 5.1.1.1 in for JINI-DMS and in Section 5.2.1.1 for JMS-DMS. However, here, we would like to compile some aspects that these technologies (i.e., JINI vs. JMS) did not address properly.

6.4.1 Potential bottleneck

JINI and JMS are both having the same centralized-model approach in terms of providing services. In JMS, the entire model is centralized with respect to the JMS provider. As a result, JMS providers are heavyweight middleware components and can become bottlenecks because the JMS specification does not address the routing of JMS messages across multiple servers or the distribution of servers to achieve load balancing. The same approach with JINI, in its lookup service and JavaSpace as its persistent database.

Although we can not remove the workload from the JMS provider or JavaSpace, we can cluster some of them to balance the load. Therefore, in critical situations, scenarios such as using clustering or grid environment becomes very useful. Each of these scenarios may bring their own problems as well, some of them having been discussed in Sections 6.4.2, 6.4.6, 6.4.7, and 6.4.8.

6.4.2 Load Balancing

In order to balance the messaging workload, as mentioned above, we can use clustering or a grid. However, load balancing has not been addressed or included directly into the JMS API, and always been taken care of by the JMS provider. JINI is suffering from the same problem as it should be implemented in a separate modules.

6.4.3 User-defined messaging

JMS has a number of different message types (e.g., Object, stream, text, map, etc.), so in each scenario we should choose the one is the most relevant one. Even though it consists of different types, it is not open to accept any new user-defined message types. For example in our scenario we use Object message which accepts any kind of Serializable Objects, and we enable different signature for each of our internal message type. This would not be the case in JINI as it works with tuple-based messaging, which is a wrapper written in Java on top of each message, so we can store them in any type, but at most it should be a Serializable Java Object like JMS.

6.4.4 Security

JMS does not specify an API for controlling the privacy and integrity of messages. JMS providers (i.e., Jboss Application Server) are responsible to monitor all incoming and outgoing messages through their authentication services. This would be the same

problem for JINI. Therefore, in order to address authentication, one should consider it in the implementation rather than having such a thing enabled in JINI specifications.

6.4.5 Communication mode

Event communication in JINI is built on top of synchronous communication (Java RMI), so the same restrictions that limit scalability and efficiency in RMI apply to JINI. However, by default, JMS supports both synchronous and asynchronous communication.

6.4.6 Internet

According to [16], JINI is usually used only within the Local Area Network (LAN), as opposed to as an internet-wide/grid-based structure like JMS, WSDL [74], or SOAP [94]. The reason for using JINI mostly in LAN is its time-consuming lookup service, which makes no sense in such environments. However, they also referred to some investigations on using the hierarchical chain of lookup services over a grid infrastructure where a lookup service is registered with another one. Thus the overhead of traversing down this chain of services would definitely cause considerable overhead to the system as each step in the traversal will require a series of full JINI lookup service and acquisition. By having the real-life escalated environment with many remote and local nodes and services, JINI would not be a very good solution for interactions except inside LAN.

6.4.7 Restricted Networks

According to [16], some routers on the internet do not support routing of multicast packets for a variety of reasons. Also, some organizations are not willing to open their firewalls for multi-casting to avoid any security problem. Similarly, a LAN divided

into subnets, may disable multicast traffic across the subnets to avoid unnecessary traffic that may result in performance degradation. This blocking of multicast traffic across subnets prohibits the use of JINI in such an environment. As both of these two technologies need the Internet Protocol (IP) address of their central dispatcher (i.e., JBoss Application Server or JavaSpace), they are restricted to work only in a network infrastructure where the IP address of that central node should not change in any startup of their hosting machine and be accessible by other nodes by their IP address.

6.4.8 Flexibility

In order to find services, JMS only requires a valid IP address, whereas JINI lookup service requires the exact full name of a service (e.g., GIPSY). However, JINI is not flexible in accepting some regular expressions such as (*) or (%) (e.g., *GIP*). This would reduce the flexibility of the JINI lookup service in grid or clustered environments. This applies to JMS as well where it requires exact IP address of the DMS node not only part or range of it.

6.4.9 Message Filtering

Both JMS and JINI address inquiries with a SQL-like query language. However both of them are having the subject-based filtering in a different approach. JINI applies the filtering on the message tuples, whereas in JMS, it searches through the message header. None of them enable content-based filtering yet, which reduces the usefulness of message filtering. If we require such a filtering in our environment, we have to implement our own filtering classes.

6.4.10 Language Dependability

Both of JINI and JMS are tightly integrated with the Java language. Therefore, they have the ability to work with any Java-based programming language and technology. Hence, in a larger-scale heterogeneous environment, all applications are not necessary Java-based. However, it has not been a problem in GIPSY, as most of all modules are implemented in Java, but still we are open in implementing Java Native Interfaces (JNI) [23] for the integration of our Java-based component with other components which implemented by using a different programming language.

6.4.11 Summary

In this chapter, we benchmarked and evaluated our DMS instances in different terms of QoS. Despite our limited network infrastructures, we simulated some real-life scenarios and situations. We experimented our system in facing different types and size of demands in situations where complexity increases. Some of them preformed expectedly and some not, but as GIPSY is a part of an ongoing project, we will continue to develop and expand this testing environment for more challenging and complicated situations. In the next chapter, we will wrap up this comparative study in Section 7.1.3 in addition to the conclusion of our research contributions.

Chapter 7

Conclusion and Future Work

This sky where we live is no place to lose your wings,

So love, love, and love.

Sh. Hafez - Persian Mystic and Poet (14th century).

7.1 Conclusion

In this chapter, we would like to conclude our research contributions in addition to mentioning our main research purposes which were to have a generic DMF and a comprehensive study over our DMS instances.

7.1.1 Research Contribution

- In the design chapter, we analyzed the workflow diagram of our demand-migration process in both current and future designs. We analyzed every step and found out what needed to be added, which we basically addressed in the projected design.

- After studying such designs, we analyzed different tiers and types of demands

that our system should have to perform better. We applied some of them in our current implementations, but major parts of these tiers as we explained in section 4.3 would be implemented in our future work.

7.1.2 Generic DMF

As mentioned before in this thesis, one of the main goals of this research was to study the behavior of the GIPSY demand-driven execution environment, and examine if our framework is generic enough in terms of accepting new distributed technologies as its DMS instances. After our design and implementation, we tested our framework with two instances of DMS. JMS-DMS and JINI-DMS functions quite similarly, but in different workload performs differently. One functions better with one type of demands, and the other one with another type. As expected, we proved that by having different instances of DMS, one can overcome the limitation of only one distributed technology.

7.1.3 Comprehensive Studies

During this research, we have done comprehensive studies over functionality, performance, and behavior of our DMS instances (see Chapter 6). There were cases that each of our instances performs similarly (i.e., latency, persistency, flexibility, and configurability), whereas in some cases only one of them was outstanding (i.e., availability for JMS-DMS or throughput for JINI-DMS).

There were cases (e.g., throughput) one instance (JMS-DMS) starts off the experiment very well, but as we increased the workload or complexity of the scenario, the other one's performance (JINI-DMS) was noticeably impressive.

After doing all these studies, we noticed that the combination of our studies in latency, dispatching, and throughput can give us a better picture or how our systems

can escalate in different challenging situations, which we would like to consider this combination as the scalability of our DMS.

By studying these factors, we found out that both systems perform well when it comes to latency time, but in terms of throughput, JINI-DMS stays in a much better position. However, JMS-DMS has better availability rate and dispatching time. Thus, we can see that one is stronger in some situations, and the other is having better performance for other possible situations. Considering this fact, we would like to consider both DMS instance very scalable.

However, for our final solutions for the current state of our GIPSY project, we would like to consider JMS-DMS as the one which is much more available, and performs faster in terms of establishing a connection. It has many embedded features that in contrary should be implemented in JINI-DMS, which increase the complexity and overhead of our system configurability and maintenance. The priority mechanism in JMS was easier to use and expand rather than JINI, which requires synchronized management layers to perform the same level of functionality. Implementation-wise JMS has been used widely in many enterprise and academic projects that install in very wide networks such as Internet or Grid, whereas JINI does not function at the same level in those environments. Both of them suffer from many issues mainly mentioned in section 6.4, but JMS is still an ongoing project in SUN, whereas JINI is discontinued by SUN, and is embedded inside a different project (i.e., River) in Apache. On the other side, JINI-DMS performs considerably better in terms of throughput of larger-sized demands as its MST was by far better than JMS-DMS. Having said that before that it is because of the well-performing Object storage (i.e., JavaSpace), we can easily replace this with our HSQLDB to take advantage of the same level of functionality. We can also extend this by replacing JavaSpace with a better option (see section 7.2.1).

7.2 Future Work

We have listed the following short and long-term future work.

7.2.1 Short-term Future Work

- Implement an efficient management tier to control the entire process of demand migration.
- Improve the demand store in DMS by including different features such as garbage collector, queue browser and etc.
- Implement other instances of DMS by using different technologies to experiment other features of distributed systems inside GIPSY DMF.
- Find an optimum persistent database to store our demands which can escalate in different situations and fit in all of our DMS instances as the generic standard solution. We can start this study by replacing the HSQLDB with JavaSpace or other successful solutions. Currently each of these two DMS are using their own persistent storages, which require two different technical mechanism and approaches. In order to improve the integration among GIPSY modules, we can implement persistency by using some standard and well-know techniques such as Hibernate [27]. It providers both high and low level control over the stored data (i.e., GIPSY Demands).
- Implement a flow-control layer as one of submodules of the monitor component in DMS to distribute the workload of communication channels and distribute them before exceeding their MST.
- Study the impact and overhead of storing demands and their results in the Demand Store of the DMS for any further access of that specific demand instead

of re-migration and re-computation.

7.2.2 Long-term Future Work

- Implement the entire DMS and execution nodes by applying the multi-tier architecture
- Include different modules of management, security, and authentication on top of each tiers to monitor the security issues of the system
- Include different features and characteristic of Autonomic Computing into the DMF and subsequently DMS instances.
- Direct our DMS and DMF toward implementing the Service-Oriented Architecture (SOA) [42] to provide a service-based computing environment for our GIPSY services. So far, we have implemented many of the required characters of SOA in our DMS, but we should step forward to that extent.

Bibliography

[1] http://www.sun.com/, (viewed on July 2008).

[2] http://www.oracle.com, (viewed on July 2008).

[3] http://www.bea.com, (viewed on July 2008).

[4] http://www.microsoft.com, (viewed on July 2008).

[5] BEA WebLogic Server©10 Data Sheet. http://www.oracle.com, (viewed on July 2008).

[6] IBM Ltd. http://www.ibm.com/ca/, (viewed on July 2008).

[7] JBoss Application Server Guide. `http://www.jboss.org/products/jbossas` (viewed on August 2008).

[8] JORAM: Java Open Reliable Asynchronous Messaging-Object Web. http://joram.objectweb.org (viewed on August 2008).

[9] Mantaray Guide. http://java-source.net/open-source/jms/mantaray, (viewed on July 2008).

[10] Oracle9i Application Developer's Guide - Advanced Queuing Release 2 (9.2). http://download.oracle.com/docs/cd/B10500-01/appdev.920/a96587/qintro.htm, (viewed on July 2008).

[11] Tibco Rendez-Vous. (viewed on July 2008) http://www.tibco.com/resources/software/messaging/rendezvous-ds.pdf.

[12] GlassFish Quick Start Guide, 2008. (viewed on July 2008) https://glassfish.dev.java.net/downloads/quickstart.

[13] IBM WebSphere MQ : Universal messaging backbone for SOA, Version 7.0 Data Sheet, 2008. ftp://ftp.software.ibm.com/software/integration/wmq/WebSphere-MQ-V7-Data-Sheet.pdf, (viewed on July 2008).

[14] A. H. Pourteymour and E. Vassev and J. Paquet. Towards a New Demand-Driven Message-Oriented Middleware in GIPSY. In *Proceedings of the Parallel and Distributed Processing Techniques and Applications (PDPTA) 2007, Las Vegas, USA*, June 2007.

[15] A. H. Pourteymour and E. Vassev and J. Paquet. Design and Implementation of Demand Migration Systems in GIPSY. In *Proceedings of the Parallel and Distributed Processing Techniques and Applications (PDPTA) 2008, Las Vegas, USA*, July 2008.

[16] A. Theneyan and P. Mehrotra and M. Zubair. Enhancing JINI for use across non-multicastable network. In *Proceedings of the ACM SIGOPS Operating Systems Review, Banff, Alberta, Canada*, April 2001.

[17] C. Amza and A.L. Cox. TreadMarks: Shared Memory Computing on networks of Workstations. *IEEE Computer*, 29:18–28, February 1996.

[18] B. Angered and A. Erlacher. Loosely coupled communication and coordination in next-generation java middleware, 2007. (viewed on July 2008) `http://today.java.net/lpt/a/197`.

[19] Apache. River Guideline, 2008. (viewed on July 2008) http://incubator.apache.org/river/RIVER/index.html.

[20] E.A. Ashcroft. Dataflow and eduction: data-driven and demand-driven distributed computation. *Springer-Verlag New York, Inc. , Current trends in concurrency. Overviews and tutorials*, pages 1–50, 1986.

[21] B. Lu. *Framework for the General Eduction Engine (GEE) in the GIPSY.* PhD thesis, Department of Computer Science and Software Engineering, Concordia University, 2004.

[22] B. Lu and P. Grogono and J. Paquet. Distributed execution of multidimensional programming languages. In *Proceedings 15th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2003)*, volume 1, pages 284–289. International Association of Science and Technology for Development, November 2003.

[23] B. Sterns. *The Java Native Interface (JNI).* Sun Microsystems, Inc., 2001-2005. `http://java.sun.com/developer/onlineTraining/Programming/JDCBook/jni.html` (viewed on July 2008).

[24] B. Ban. JGroups : A Toolkit for Reliable Multicast Communication. Technical report, 2008. http://www.jgroups.org/javagroupsnew/docs/ug.html, (viewed on July 2008).

[25] S. Boyd. *Pro MSMQ: Microsoft Message Queue Programming.* Apress, 2004. ISBN (1590593464).

[26] F. Buschmann, R. Meunier, H. Rohnert, and P. Sommerlad. *Pattern-Oriented Software Architecture.* Wiley, 1996. ISBN (978-0471958697).

[27] C. Bauer and G. King. *Java Persistence with Hibernate.* Manning Publications, 2006. ISBN (1932394885).

[28] R. G. G. Cattell and D. K. Barry. *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann Publishers, 2000. ISBN (0201633612).

[29] P. Chan and R. Lee. *The Java(TM) Class Libraries Poster: Java 2 Platform Standard Edition 5.0.* Prentice Hall PTR, 2004. ISBN (0321304780).

[30] S. Chen and P. Greenfield. QoS Evaluation of JMS: An Empirical Approach. In *Proceedings of the 37th Hawaii International Conference on System Sciences,* 2004.

[31] Apache Community. Activemq : Getting started. Technical report, July 2008. http://activemq.apache.org/getting-started.html, (viewed on August 2008).

[32] Jboss Community. Jboss enterprise application platform, rt(364475). Technical report, Red Hat and JBoss Enterprise Middleware, April 2007. http://www.jboss.com/pdf/jb-ent-app-platform-04-07.pdf, (viewed on July 2008).

[33] C. Crist. HermesJMS Installation Guide. Technical report, Hermes.

[34] J. Crowcroft, S. Hand, R. Mortier, T. Roscoe, and A. Warfield. QoS's Downfall: At the bottom, or not at all. *ACM Sigcomm RIPQOS Workshop,* August 2003. `http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/MDB.html, (viewedonJuly2008).`

[35] F. Cummins. *Enterprise Integration : An Architecture for Enterprise Application and System Integration.* Wiley Computer Publishing, OMG Press, 2002. ISBN (0471400106).

[36] J. Day. The (un)revised OSI reference model. *SIGCOMM Comput. Commun. Rev.*, 25(5):39–55, 1995.

[37] C. Dodd. *Intensional Programming I, chapter Rank analysis in the GLU compiler, pages 76-82.* World Scientific, Singapore, 1996.

[38] E. I. Vassev. General Architecture for Demand Migration in the GIPSY Demand-Driven Execution Engine. Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, June 2005. ISBN 0494102969.

[39] E. Sinderson and V. Magapu and R. Mak. Middleware and web services for the collaborative information portal of nasa's mars exploration rovers mission. In *Middleware '04: Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, pages 1–17, New York, NY, USA, 2004. Springer-Verlag New York, Inc.

[40] E. Vassev and J. Paquet. A General Architecture for Demand Migration in a Demand-Driven Execution Engine in a Heterogeneous and Distributed Environment. In *Proceedings of the 3rd Annual Communication Networks and Services Research Conference (CNSR 2005), Halifax, Nova Scotia*, pages 176–182. IEEE, May 2005.

[41] E. Vassev and J. Paquet. A Generic Framework for Migrating Demands in the GIPSY's Demand-Driven Execution Engine. In *Proceedings of the 2005 International Conference on Programming Languages and Compilers (PLC 2005), Las Vegas, USA*, pages 29–35. CSREA Press, June 2005.

[42] T. Erl. *Service-Oriented Architecture (SOA): Concepts, Technology, and Design.* Prentice Hall PTR, 2005. ISBN (0131858580).

[43] P. Th. Eugster, P.A. Felber, and R.Guerraoui. Too many faces of publish subscribe. *ACM computing surveys*, 2(2):114–131, 2003.

[44] F. Sommers. Activatable Jini services, Part 1: Implement RMI activation. *JavaWorld.com*, 2000. http://www.javaworld.com/javaworld/jw-09-2000/jw-0915-jinirmi.html, (viewed on July 2008).

[45] T. Faison. *Event-Based Programming Taking Events to the Limit.* Apress, 2006. ISBN (0321304780).

[46] J. Farley. *Java Distributed Computing.* O'Reilly, 1998. ISBN (1565922069).

[47] A. A. Faustini and W. W. Wadge. An eductive interpreter for the language Lucid. *SIGPLAN Not.*, 22(7):86–91, 1987.

[48] S. Ferg. *Event-Driven Programming: Introduction.* 2006.

[49] R. Flenner. *JINI and JavaSpaces Application Development.* Sams, 2001.

[50] G. Hohpe and B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions.* Addison-Wesley Professional, 2003. ISBN (0321200683).

[51] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java Concurrency in Practice.* Addison-Wesley Professional, 2006. ISBN (0321349601).

[52] M. Hapner, R. Burridge, R. Sharma, J. Fialli, and K. Stout. *Java(TM) Message Service API Tutorial and Reference.* Prentice Hall PTR, 2002. ISBN (0201784726).

[53] G. Hohpe. An Asynchronous World. *Dr.Dobb's magazine*, 2003. `http://www.ddj.com/architect/184415001,(viewedonJuly2008).`

[54] D. Hunter. *Beginning XML, 4th Edition (Programmer to Programmer).* Wrox Press, 2007. ISBN (0470114878).

[55] J. Newmarch. *Foundations of JINI 2 Programming.* Apress; 1 edition, 2006. ISBN (1590597168).

[56] J. Paquet. *Scientific Intensional Programming.* PhD thesis, Department of Computer Science, Laval University, Sainte-Foy, Canada, 1999.

[57] J. Paquet and S. A. Mokhov and X. Tong. Engineering Context Calculus in the GIPSY Environment. In *Proceedings of the 1st IEEE International Workshop on Software Engineering for Context Aware Systems and Applications (SECASA 2008)*, Turku, Finland, 2008. To appear, `http://conferences.computer.org/compsac/2008/workshops/SECASA2008.html`.

[58] J. Plaice and J. Paquet. Introduction to intensional programming. In *Intensional Programming I, World Scientific, Singapore*, pages 1–14, 1996.

[59] J. Tang and W. Tong and J. Ding and L. Cai. MOM-G: Message-Oriented Middleware on Grid Environment Based on OGSA. In *ICCNMC '03: Proceedings of the 2003 International Conference on Computer Networks and Mobile Computing*, page 424, Washington, DC, USA, 2003. IEEE Computer Society.

[60] JINI Community. *JINI Network Technology.* Sun Microsystems, Inc., September 2007. `http://java.sun.com/developer/products/jini/index.jsp` (viewed on July 2008).

[61] K. Hasse. *Java Message Service API Tutorial.* Sun Microsystems, Inc., 2002.

[62] K. Mani Chandy and S. Ramo. Event driven architecture. Technical report, California Institute of Technology, 2007.

[63] K. Wan and V. Alagar and J. Paquet. A Context theory for Intensional Programming. In *Workshop on Context Representation and Reasoning (CRR05), Paris, France*, July 2005.

[64] K. Wan and V. Alagar and J. Paquet. Lucx: Lucid Enriched with Context. In *Proceedings of the 2005 International Conference on Programming Languages and Compilers (PLC 2005), Las Vegas, USA*, pages 48–14. CSREA Press, June 2005.

[65] A. Kupsys and R. Ekwall. Architectural Issues of JMS Compliant Group Communication. In *Proceedings of the 4th IEEE International Symposium on Network Computing and Applications (NCA'05)*, July 2005.

[66] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development.* Pearson Education, third edition, April 2005. ISBN: 0131489062.

[67] N. Lynch. *Distributed Algorithms.* Kaufmann Publishers Inc., 1996. ISBN (1558603484).

[68] M. Musolesi and C. Mascolo and S. Hailes. EMMA: Epidemic Messaging Middleware for Ad hoc networks. *Springer-Verlag London Limited*, pages 28–36, 2005.

[69] M. Menth, R. Henjes, S. Gehrsitz, and C. Zepfel. Throughput Comparison of Professional JMS Servers. Technical report, University of Würzburg, Institute of Computer Science, March 2006. Technical Report No. 380.

[70] S. A. Mokhov. Towards hybrid intensional programming with JLucid, Objective Lucid, and General Imperative Compiler Framework in the GIPSY. Master's

thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, October 2005. ISBN (0494102934).

[71] G. Muhl. *Distributed Event-Based Systems.* Springer, 2006. ISBN (3540326510).

[72] N. Richards and S. Griffith. *JBoss: A Developer's Notebook (Developers Notebook).* O'Reilly Media, Inc., 2005. ISBN (0596100078).

[73] N. S. Papaspyrou and I. T. Kassios. GLU# embedded in C++: A marriage between multidimensional and object-oriented programming. *Softw., Pract., Exper.*, 34(7):609–630, 2004.

[74] O. Zimmermann and M. R. Tomlinson and S. Peuser. *Perspectives on Web Services: Applying SOAP, WSDL and UDDI to Real-World Projects.* Springer, 2005. ISBN (3540009140).

[75] OpenJMS Community. Using OpenJMS Guide. Technical report, March 2008. http://openjms.sourceforge.net/usersguide/using.html, (viewed on July 2008).

[76] C. O'Ryan. *Empirical evaluation of design patterns for publisher-subscriber distributed systems.* PhD thesis, Department of Computer Science, University of California, Irvine, 2002.

[77] P. Bishop and N. Warren. JINI-like discovery for RMI. *JavaWorld.com*, 29, November 2001. http://www.javaworld.com/javaworld/jw-11-2001/jw-1121-jinirmi.html?page=4, (viewed on July 2008).

[78] Joey Paquet and Peter Kropf. The GIPSY Architecture. In *Proceedings of Distributed Computing on the Web, Quebec City, Canada*, 2000.

[79] Joey 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.

[80] A. Puder, K. Romer, and F. Pilhofer. *Distributed System Architecture : A Middleware Approach.* Morgan Kaufmann Publisher, 2006. ISBN (978-1-55860-648-74).

[81] R. Pyarali, D.Schmidt, and R.K. Cytron. Techniques for enhancing real-time CORBA quality of service. In *Proceedings of the IEEE Vol.91, NO.7*, July 2003.

[82] R. Baldoni and R. Beraldi and V. Quema and L. Querzoni and S. T. Piergiovanni. TERA: Topic-based Event Routing for Peer-to-Peer Architectures. In *Inaugural Conference on Distributed Event-Based Systems (DEBS07), ACM press, 2-13.* ACM, 6 2007.

[83] R. Henjes and M. Menth and C. Zepfel. Throughput performance of java messaging services using webspheremq. In *ICDCSW '06: Proceedings of the 26th IEEE International Conference Workshops on Distributed Computing Systems*, page 26, Washington, DC, USA, 2006. IEEE Computer Society.

[84] S. Rooney, D. Bauer, and P. Scotton. Distributed Messaging using Meta Channels and Message Bins. In *Proceedings of the 9th IFIP/IEEE International Symposium on Integrated Network Management*, pages 99–105. IBM Research, Zurich Research Laboratory, May 2005.

[85] S. Mokhov and J. Paquet. General Imperative Compiler Framework within the GIPSY. In *Proceedings of the 2005 International Conference on Programming Languages and Compilers (PLC 2005), Las Vegas, USA*, pages 36–42. CSREA Press, June 2005.

[86] S. Tai and I. Rouvellou. Strategies for Integrating Messaging and Distributed Object Transactions. In *Proceedings of Middleware 2000,*, pages 308–330. Springer-Verlag Berlin Heidelberg, LNCS 1795, 2000.

[87] B. Shannon. Java 2 Enterprise Edition specification, Edition 1.4. Technical report, Microsystems Inc., April 2003.

[88] Krissoft Solutions. JMS Performance Comparison : Performance Comparison for Publish Subscribe Messaging. Technical report, Krissoft Solutions, 2004.

[89] Sun Microsystems. *The J2EE 1.4 Tutorial: Types Supported by JAX-RPC.* Sun Microsystems, Inc., 2006. `http://java.sun.com/j2ee/1.4/docs/tutorial/doc/JAXRPC4.html#wp130550`.

[90] Sun Microsystems. *Java Message Service (JMS).* Sun Microsystems, Inc., September 2007. `http://java.sun.com/products/jms/`.

[91] Sun Microsystems, Inc. Java Native and Directory Interface (JNDI). http://java.sun.com/products/jndi, (viewed on July 2008).

[92] A. S. Tanebaum and M. Van Steen. *Distributed Systems: Principals and Paradigms.* Prentice Hall, 2006. Second Edition ISBN (0-13-239227-5).

[93] The HSQLDB Development Group. HSQLDB - Lightweight 100% Java SQL Database Engine v.1.8.0.4. hsqldb.org, 2006. `http://hsqldb.org/`, (viewed on August 2008).

[94] W. B. Brogden and B. Brogden. *SOAP Programming with Java.* Sybex Inc., 2002. ISBN (0782129285).

[95] Kaiyu Wan. *Lucx: Lucid Enriched with Context.* PhD thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2006.

[96] Ann Wollrath and Jim Waldo. *Java RMI Tutorial.* Sun Microsystems, Inc., 1995-2005. `http://java.sun.com/docs/books/tutorial/rmi/index.html`.

[97] B. Woolf. Event-driven architecture and service-oriented architecture. Technical report, IBM Corporation, 2006.

[98] E. Yoneki and J. Bacon. Pronto: Messaging Middleware over Wireless Networks. In *Proceedings of the ACM/IFIP/USENIX International Middleware Conferenc*, 2003.

Index