

SCALABILITY EVALUATION OF THE GIPSY RUNTIME
SYSTEM

YI JI

A THESIS
IN
THE DEPARTMENT
OF
COMPUTER SCIENCE AND SOFTWARE ENGINEERING

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF APPLIED SCIENCE (SOFTWARE ENGINEERING) AT
CONCORDIA UNIVERSITY
MONTRÉAL, QUÉBEC, CANADA

MARCH 2011

© YI JI, 2011

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: Yi Ji

Entitled: Scalability Evaluation of the GIPSY Runtime System

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Software Engineering)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Chair
Dr. Brigitte Jaumard

_____ Examiner
Dr. Todd Eavis

_____ Examiner
Dr. Olga Ormandjieva

_____ Supervisor
Dr. Joey Paquet

Approved by

Chair of Department or Graduate Program Director

_____ 20 _____

Dr. Robin A. L. Drew, Dean
Faculty of Engineering and Computer Science

Abstract

Scalability Evaluation of the GIPSY Runtime System

Yi Ji

Intensional programming is a declarative programming paradigm that is suitable for scientific programming since it allows natural expression of equations regarding multidimensional objects or concepts evolving in a multidimensional context so that the simplicity of these equations are kept.

The General Intensional Programming System (GIPSY) project aims at providing a software platform for the long-term investigation of intensional programming. The GIPSY consists of a flexible compiler and a scalable runtime system, where the compiler translates any flavor of intensional program into source-language independent runtime resources, and the runtime system uses the runtime resources to execute the program in a demand-driven and distributed manner, i.e. computation requirements are wrapped into demands and are distributed among networked computers, so that the computations can be executed distributively and concurrently to shorten their overall computation time.

The multi-tier architecture adopted for the GIPSY runtime system is for research goals such as scalability. It consists of the Demand Generator Tier that generates demands, the Demand Store Tier that stores and dispatches demands, as well as the Demand Worker Tier that computes demands. All the tiers are allocated in registered computers called the GIPSY nodes, and all the GIPSY nodes and tiers are under the management of the General Manager Tier, with which new nodes can be registered and new tiers can be allocated at runtime to deal with increasing workload. This thesis covers the development of the scalable GIPSY runtime system using the multi-tier architecture, and presents the assessment of the scalability of the developed GIPSY runtime system.

Acknowledgments

I would like to express my sincere gratitude to my supervisor Dr. Joey Paquet for giving me the opportunity to commence my degree and research, and for his patience, caring guidance and thorough support during my graduate study.

I would also like to thank all the GIPSY team members, specifically, Bin Han and Serguei A. Mokhov, for outstanding teamwork.

Finally, I would like to thank my dearest parents and my beloved fiancée for their unconditional love, continuous encouragement and everlasting support.

This work has been sponsored by the Natural Sciences and Engineering Research Council of Canada and the Faculty of Engineering and Computer Science of Concordia University, Montreal, Quebec, Canada.

Contents

List of Figures	viii
List of Tables	x
1 Introduction	1
1.1 General Intensional Programming System	1
1.1.1 Intensional Programming and GIPSY	2
1.1.2 The GIPSY Framework	3
1.2 Previous Work on the GIPSY Runtime System	5
1.2.1 Pre-Multi-Tier Era	5
1.2.2 The Multi-Tier Architecture	7
1.3 Scalability Evaluation	12
1.3.1 Definition of Scalability	12
1.3.2 Scalability Metrics	16
1.3.3 Scalability Evaluation of the GIPSY Runtime System	19
1.4 Thesis Overview	21
1.4.1 Scope	21
1.4.2 Problem Statement	22
1.4.3 Contributions	23
1.4.4 Outline	24
2 Development of the GIPSY Runtime System	25
2.1 Requirement Analysis	25

2.2	Design	31
2.2.1	Configuration System	31
2.2.2	GIPSY Instance Bootstrap Process	33
2.2.3	GIPSY Node Registration	34
2.2.4	GIPSY Tier Allocation and Deallocation	36
2.3	Implementation	39
2.3.1	Refactoring the Jini DMS and the JMS DMS	39
2.3.2	Implementation of the Jini and the JMS DSTs	40
2.3.3	Implementation of the GMT and the GIPSY Node	44
2.4	Tests	49
2.4.1	GIPSY Instance Bootstrap and GIPSY Node Registration	49
2.4.2	GIPSY Tier Allocation and Deallocation	52
2.5	Summary	56
3	Scalability Evaluation	58
3.1	Overview	58
3.2	Space Scalability	60
3.2.1	Space Scalability Test: Demand Storage	63
3.2.2	Space Scalability Test: Maximum Demand Size	67
3.2.3	Summary	70
3.3	Space-Time Scalability	70
3.3.1	Space-Time Scalability Test: Memory Impact	71
3.3.2	Space-Time Scalability Test: Signature Matching Speed	74
3.3.3	Summary	79
3.4	Structural Scalability	79
3.4.1	Structural Scalability Test: GMT	80
3.4.2	Structural Scalability Test: DST	82
3.4.3	Summary	84
3.5	Load Scalability	85
3.5.1	Load Scalability Test: All the Tiers in One GIPSY Node	86

3.5.2	Load Scalability Test: DWTs Scaled Out in One GIPSY Node	88
3.5.3	Load Scalability Test: DWTs Scaled Out in Multiple GIPSY Nodes	90
3.5.4	Load Scalability Test: Two DGT Nodes for the JMS DST . .	93
3.5.5	Summary	95
3.6	Conclusion	96
4	Conclusions and Future Work	97
4.1	Conclusions	97
4.2	Limitations and Future Work	98
	Bibliography	101

List of Figures

1	Early GEE architecture	6
2	GIPSY Demand Migration System	6
3	Demand migration and state transition	8
4	Procedural demand migration among the DGT, the DST, and the DWT	11
5	Example of GIPSY instances reproduced from [1]	12
6	The multi-tier packages of the GIPSY runtime system	26
7	The relations among the classes in the multi-tier packages	28
8	GMT use cases	29
9	Configuration files of the GIPSY runtime system	32
10	Configuration property naming structure	33
11	GIPSYNode bootstrap process	34
12	GIPSYNode registration process	35
13	GIPSY tier allocation process	37
14	GIPSY tier deallocation process	38
15	Jini DST wrapper and the JMS DST wrapper	41
16	TAs and the TA factory	43
17	The system demands required by the GMT	46
18	The GMT wrapper class	46
19	The node-registration sequence diagram	47
20	The tier-allocation sequence diagram	48
21	The tier-deallocation sequence diagram	48
22	Test scenario for GIPSY instance bootstrap and node registration . .	49
23	Screenshots of the GMT console and the two GIPSY Node consoles .	51

24	Test scenario for GIPSY tier allocation and deallocation	52
25	GMT console output for tier allocation and deallocation	54
26	The four DSTs allocated	55
27	The console output of Node 0	56
28	The console output of Node 1	57
29	JVM heap generations	62
30	Demand storage test deployment	63
31	Maximum total demand size versus heap size	66
32	Maximum demand size for 256 MB heap	69
33	Space-time scalability test deployment: memory impact	71
34	Response time versus demand storage	72
35	Space-time scalability test deployment: signature matching speed . .	75
36	Demand sending time versus the amount of demands stored in the DST	76
37	Average demand sending time versus the amount of demands stored in the DST	76
38	Result reading time versus the amount of demands stored in the DST	77
39	Average result reading time versus the amount of demands stored in the DST	78
40	Structural scalability test deployment: GMT	81
41	Structural scalability test deployment: DST	82
42	DST connections and threads versus stack sizes	84
43	Load scalability test deployment: all the tiers in one GIPSY node . .	87
44	Load scalability test result: all the tiers in one node	88
45	Load scalability test deployment: DWTs scaled out in one GIPSY node	89
46	Load scalability test result: DSTs scaled out in one GIPSY node . . .	90
47	Load scalability test deployment: DWTs scaled out in multiple GIPSY nodes	91
48	Load scalability test result: DSTs scaled out in multiple GIPSY nodes	92
49	Load scalability test deployment: two DGT nodes for the JMS DST .	93
50	Load scalability test result: two DGT nodes for the JMS DST	95

List of Tables

1	Computer hardware and operating system environment	50
2	Hardware and operating system environment	64
3	The 12 DST profiles	65
4	Demand storage test result	66
5	Maximum demand size test result	68
6	DST maximum connection test result	83

Chapter 1

Introduction

The General Intensional Programming System (GIPSY) project aims at providing a software platform for the long-term investigation of intensional programming [1]. The GIPSY basically consists of a flexible compiler and a scalable runtime system for the compilation and execution of intensional programs, where the compiler translates any flavor of intensional programs into source-language independent runtime resources, and the runtime system uses the runtime resources to execute the program in a demand-driven and distributed manner, i.e. computation requirements are wrapped into demands and are distributed among networked computers, so that the required computations can be executed distributively and concurrently to shorten their overall computation time.

Since this thesis concerns the development of the runtime system for the GIPSY project as well the assessment of the scalability of the runtime system, this chapter introduces the necessary background, previous work and limitations regarding the GIPSY runtime system and the scalability evaluation methodologies, and delivers the scope, problem statement, contributions and the outline of this thesis.

1.1 General Intensional Programming System

The General Intensional Programming System (GIPSY) project is for the investigation of the intensional programming model, and is an ongoing project

directed by Dr. Joey Paquet in the Computer Science & Software Engineering Department at Concordia University in Montreal, Quebec.

1.1.1 Intensional Programming and GIPSY

Intensional programming is a declarative programming paradigm that is suitable for scientific programming, since it allows natural expression of scientific equations regarding multidimensional objects or concepts evolving in a multidimensional context so that the simplicity of these scientific equations are kept [2]. It has found applications in 3D spreadsheet [3, 4], parallel programming [5], real-time systems [6, 7], software version control [8, 9] and database systems [10].

Basically the intensional programming model allows the programmers to directly manipulate their *intensions* [2]. When solving complex problems of multidimensional nature such as plasma physics or tensor problems, the scientific equations can be expressed naturally in an intensional programming language. In contrast, the solutions written in conventional programming languages such as C, FORTRAN and Java do not reflect the simplicity and readability of the original scientific equations, and the solutions provided by mathematical programming languages such as Matlab and Maple are ad-hoc software packages such as the Tensor package or toolbox that are specific to the particular sets of problems only, lacking the simplicity and generalization in directly defining the equations naturally via the intensional programming model [2].

An example of the intensional programming languages is the evolving Lucid family [11]. Initially Lucid was a data-flow programming language that used only the time dimension [12, 13, 14, 15]. Then in 1996 it was evolved to Indexical Lucid that supported multiple dimensions and Granular Lucid (GLU) that supported hybrid programming in C or FORTRAN and Indexical Lucid [16, 17]. In 1999 Dr. Paquet proposed the General Intentional Programming Language (GIPL) into which all the Lucid dialects can be translated [2], hence the name GIPL. After this generalization, JLucid that embedded Java into Indexical Lucid [18, 19] and Object-Oriented Lucid Intensional Programming Language that combined features of both

Java and Lucid [20, 21, 22] were proposed, enriching the Lucid family with hybrid intensional programming.

An example showing the advantage of Lucid is the Lucid solution to the *Hamming Problem* (after Richard Hamming). The *Hamming problem* requires generating an infinite sequence of ascending and non-repeating integers beginning from 1, where the numbers after 1 are the multiples of the numbers 2, 3, or 5. The imperative solutions of the *Hamming* problem usually involves multiple loops and sorting, whereas in Lucid a trivial and easy-understanding solution is provided [11, 23]:

```
H
where
  H = 1 fby merge ( merge ( 2*H , 3*H ) , 5*H );
  merge ( x , y ) = if ( xx <= yy ) then xx else yy
  where
    xx = x upon ( xx <= yy );
    yy = y upon ( yy <= xx );
  end;
end;
```

Listing 1.1: An example of a Lucid program

To investigate and demonstrate the possibilities and effectiveness of the intensional programming model as well as to cope with changes and general needs, the GIPSY project proposes a software platform to compile and execute programs written in any flavors of Lucid including hybrid intensional programming languages.

1.1.2 The GIPSY Framework

The GIPSY framework consists of three main subsystems: a flexible compiler called the General Intensional Programming Compiler (GIPC), a language-independent runtime system called the General Education Engine (GEE) and a component called the Runtime Interactive Programming Environment (RIPE) that provides visual user interfaces to allow user interaction with the runtime system.

Basically the compiler, GIPC, compiles any GIPSY program into an intermediate representation. A typical GIPSY program is a hybrid program consisting of the Lucid (intensional) part written in various dialects of Lucid as well as the procedural part written in various programming languages such as C, FORTRAN and Java, where the

Lucid part calls the procedures or methods defined in the procedural part [24, 25, 18, 19, 20, 21, 22]. Given such a GIPSY program, the compiler first translates the GIPSY program into a source-language independent General Intensional Programming Language (GIPL) program using predefined translation rules, then generates source-language independent Generic Education Engine Resources (GEER) based on the GIPL program. More specifically:

- **Generic Education Engine Resources (GEER):** an intermediate representation generated by the compiler GIPC containing runtime resources for the execution of a GIPSY program [19]. Since a GEER is generated from a source-language independent GIPL program that is generated from a hybrid intensional program, the GEER is also source-language independent, i.e. generic as its name implies, and contains all the GIPSY program identifiers, including both the Lucid identifiers and the procedural identifiers embedded in the hybrid program, as well as typing information, dimensionality information, the abstract syntax tree (AST) of the source-language independent GIPL program, and a dictionary of the procedures called by the Lucid parts of the original hybrid program. This GEER is then passed to the General Education Engine (GEE), i.e. the runtime system, for the program execution. Since the GEER is source-language independent, the runtime system is also source-language independent.

When the GEE, i.e. the runtime system, receives a GEER generated by the compiler, it executes the program using a demand-driven education model. A *demand* is a request asking for the value of a certain identifier defined in a GIPSY program, and the identifier is usually defined as an expression using other identifiers and an underlying algebra within a specific multidimensional context space [1]. In the demand-driven education model, an initial demand requesting the value of a certain identifier is generated, and to consume this demand, new demands are generated to request the values of the identifiers constituting the expression defining the initial identifier, and similarly these demands further generate new demands until eventually some of the demands are evaluated and propagated back in the chain of demands,

so that the identifiers whose values depend on them can be evaluated in turn, and eventually the initial identifier is evaluated and returned [1]. This demand-driven education model naturally supports distributed execution of intensional programs [17].

The development of the demand-driven GIPSY runtime system has gone through several stages via the previous work introduced in Section 1.2.1.

1.2 Previous Work on the GIPSY Runtime System

1.2.1 Pre-Multi-Tier Era

The early architectural design of GEE, i.e. the runtime system, specified that the GEE generally had three parts: the Intensional Demand Propagator (IDP) that generates demands, the Intensional Value Warehouse (IVW) that caches the values of computed demands, and the Ripe Function Executor (RFE), i.e. the worker that does functional computation as shown in Figure 1 [26]. In this architecture, the IDP generates demands according to the intensional data dependence structure generated by the compiler. If a demand generated by the IDP requires some functional computation, it is either calculated by a local worker or sent to a remote worker, decided by the Demand Dispatcher of the IDP, and when the demand is computed, its value is returned to the IDP and put into the IVW for future reuse. The IVW consists of a warehouse that stores computed values and a garbage collector to remove the stored values according to certain removal algorithms. This runtime architecture was investigated and developed by Lu [27] and Tao [28].

Later, a generic Demand Migration Framework (DMF) for migrating demands in heterogeneous and distributed environment was proposed for the GIPSY runtime system [29], and a Demand Migration System (DMS) implementing the DMF using the Jini technology [30] was provided by Vassev [31]. Later, another DMS based on the JMS technology JMS [32] was provided by Pouteymour [29, 33].

Figure 2 shows a GIPSY Demand Migration System. In this figure, the GIPSY

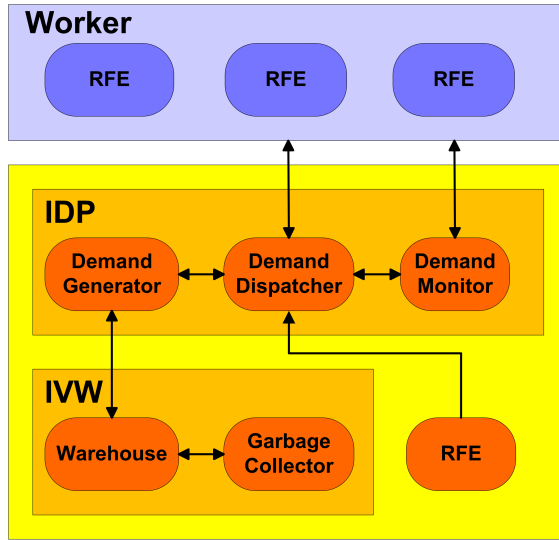


Figure 1: Early GEE architecture

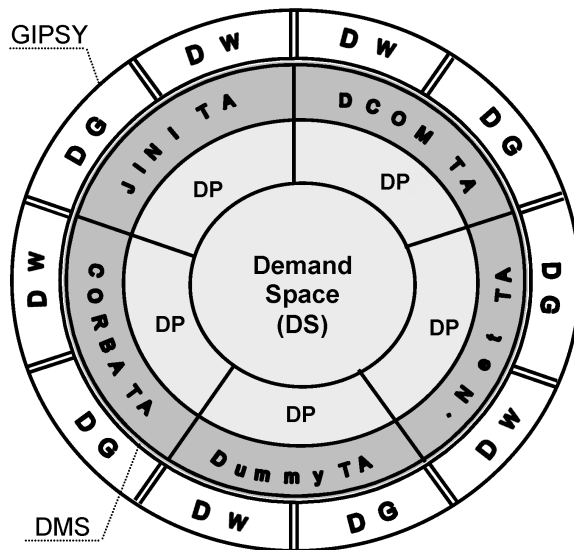


Figure 2: GIPSY Demand Migration System

runtime system consists of Demand Generators (DGs) and Demand Workers (DWs) interconnected by a DMS (the double-lined sphere surrounded by the DWs and the DGs in Figure 2). The DMS consists of a Demand Space (DS) that is similar to the previous notion of warehouse in IVW, and a set of Dispatcher Proxies (DPs) and their corresponding Transport Agents (TAs) implemented in different middleware technologies, so that the DGs and the DWs can use the TAs to connect to the DP to send and receive demands via the DS across heterogeneous and distributed environment. Similar to the notion of generator and worker in the previous architecture, the DGs generate demands and the DWs do functional computation; however, in the DMF, the DGs no longer communicate with DWs directly, instead all the demands are dispatched via the DS who also stores computed results. The dispatching is achieved by labeling demands with three different states: *pending*, *inprocess*, and *computed*. Each demand also has a signature called *demand signature* serving as the unique identifier of this demand within the DS. Figure 3 illustrates the demand migration and dispatching process: when a demand is generated by the Demand Generator and is put into the Demand Space, its state is *pending*; when the demand is picked up by a Demand Worker, its state transits to *inprocess*; after the demand is computed by the Demand Worker, its state transits to *computed* and the *computed* demand with the same signature is put back into the Demand Space so that it can be picked up by the Demand Generator.

1.2.2 The Multi-Tier Architecture

Aiming at research goals such as scalability, a multi-tier architecture was proposed and adopted for the GIPSY runtime system [1, 34], and it is the most up-to-date architecture of the GIPSY runtime system.

The multi-tier architecture preserves some of the features in the previous Demand Migration Framework, such as the three demand states *pending*, *inprocess* and *computed* as well as the demand signature, but further divides the entire runtime system into four kinds of GIPSY tiers. Each kind of GIPSY tier consists of any number of differently implemented tier instances, and each tier instance is a separate

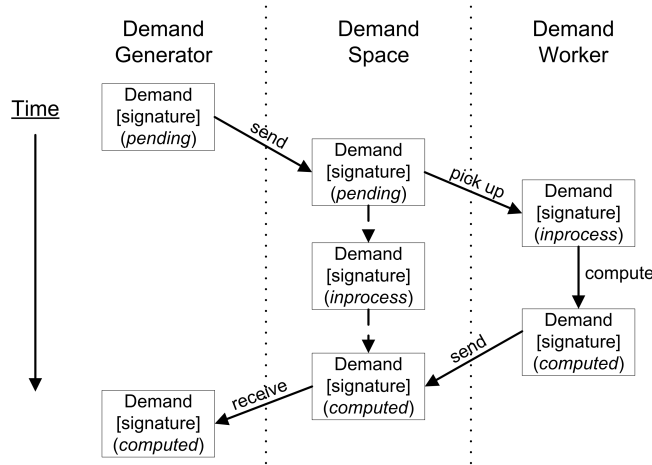


Figure 3: Demand migration and state transition

process or thread that runs within a registered computer and communicates with other tiers via demands. The registered computers are called GIPSY nodes; the creation and start of a tier instance in a GIPSY node is called the tier *allocation*, and the stop and destroy of a tier instance is called the tier *deallocation*. Since the concrete operational features of a GIPSY tier is provided by a GIPSY tier instance, unless otherwise specified, for simplicity in this thesis a *particular GIPSY tier* always denotes an *instance of the specified tier*. The detailed descriptions of the four kinds of GIPSY tiers are:

- **Demand Store Tier (DST):** a GIPSY tier whose instance serves as the middleware and cache to provide demand delivery and storage services for other tiers. Each DST exposes Transport Agents (TAs) that other tiers can obtain via a lookup service to connect to the DST either remotely or locally. The use of TA provides location transparency for the DST, and all the TAs share the same generic interface to provide implementation transparency so that new TA implementations can be easily integrated into the GIPSY runtime system without affecting the tiers who are not aware of the heterogeneity of different TA implementations. Once a demand is computed by other tiers, it is stored in the DST for future reuse. For the long-term usage, the DST is also designed to have a garbage collector to eventually remove demands out of the DST to save

space and query time.

- **Demand Generator Tier (DGT):** a GIPSY tier whose instance generates different types of demands by traversing the abstract syntax tree contained in a GEER. As described in Section 1.1.2, a GEER is a dictionary of runtime resources generated by the compiler GIPC from a hybrid intensional program consisting of the Lucid part and the procedural part. The runtime resources contained in a GEER basically include program identifiers such as Lucid identifiers and procedural identifiers, the abstract syntax tree and the procedures called by the Lucid part. When a DGT traverses the abstract syntax tree, two types of demands may be generated:

- **intensional demand:** a demand requesting the value of certain Lucid identifier contained in the Lucid part of the GIPSY program, given a certain context where the identifier is defined.
- **procedural demand:** a demand requesting the evaluation of a certain procedure defined as a function or a method in the procedural part of the GIPSY program, when a call to the function or method is encountered during the traversal of the abstract syntax tree.

A DGT also has a Local GEER Pool to store GEER instances, and an Intensional Demand Processor that uses the GEERs to process intensional demands. When receiving an initial intensional demand requesting the value of a certain Lucid identifier, the DGT checks its Local GEER Pool to retrieve the corresponding GEER. If the DGT does not have the GEER, it sends a special type of demand to the DST to request the GEER, and this type of demands is:

- **resource demand:** a demand requesting a particular GEER instance. This demand is issued by a DGT and is answered by any DGT that has the GEER. In this way GEERs are shared among multiple DGTs.

Once the DGT has the GEER, its Intensional Demand Processor traverses the abstract syntax tree contained in the GEER to generate further demands

according to the Lucid program declaration of the Lucid identifier in question. Some of these generated demands are intensional demands and some are procedural demands, depending on how the identifiers are further declared and defined. All these newly generated demands have the state *pending* and are stored in the DGT's Local Demand Store. The Local Demand Store serves as an outgoing buffer for *pending* demands, and when the Local Demand Store is empty, the DGT picks up a new intensional demand from the DST to start processing the new intensional demand. A *pending* intensional demand stored in the Local Demand Store can either be processed by a remote DGT if it is sent to the DST, or processed by the same DGT if it stays in the Local Demand Store, decided by the Demand Dispatcher of the DGT; in contrast, a *pending* procedural demand must be sent to the DST to be processed by an instance of the Demand Worker Tier described below.

- **Demand Worker Tier (DWT):** a GIPSY tier whose instance processes procedural demands by executing the procedural function calls (or methods) defined in GEERs contained in its Procedural Class Pool. When started, a DWT connects to a DST to retrieve a *pending* procedural demand, and if the DWT has the definitions of the procedures required by the procedural demand, it execute the procedures; whereas if the DWT does not have the GEER containing the definitions of the procedures required by the procedural demand, it issues a resource demand to retrieve the GEER as a DGT does in a similar situation. When the execution of the procedures is finished, the DWT returns the *computed* procedural demand to the DST, so that the *computed* demand can be picked by the DGT that generated the original procedural demand. The DWT also has a Local Demand Store to cache *computed* demands in case of DST failure. Figure 4 indicates the procedural demand migration among the DGT, the DST and the DWT.
- **General Manager Tier (GMT):** a GIPSY tier whose instance manages the GIPSY node registration and the allocation and deallocation of DGT, DST and

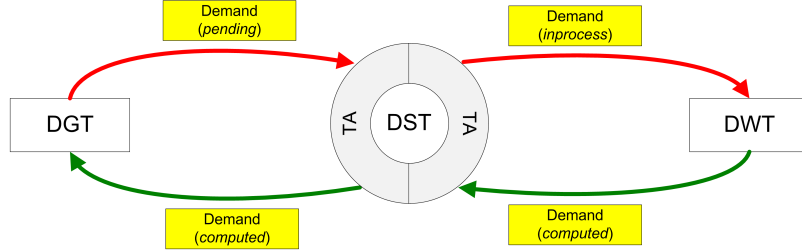


Figure 4: Procedural demand migration among the DGT, the DST, and the DWT

DWT instances. The GIPSY node registration is the process of registering a computer into the GMT so that the GMT is able to allocate GIPSY tiers in that computer when necessary. Such managerial tasks are performed via a special type of demands called:

- **system demand:** a demand for managerial tasks such as the GIPSY node registration, GIPSY tier allocation and deallocation.

Under the management of a GMT, a set of interconnected GIPSY tiers allocated in GIPSY nodes is called a **GIPSY instance**. A GIPSY instance is able to expand across the network by registering computers as GIPSY nodes and allocating GIPSY tiers in the registered GIPSY nodes. Figure 5 shows an example of three GIPSY instances identified by three different colors running in six GIPSY nodes.

The multi-tier architecture was partially implemented by Han [34, 35]. He designed the initial class diagrams and APIs of the multi-tier packages, and implemented the DGT and the DWT. His work provided a solid foundation for the multi-tier architecture and verified its feasibility. However, further development and integration is still required to deliver a deployable GIPSY instance to demonstrate the effectiveness of the multi-tier architecture towards its research goals such as scalability, which requires the development of the DSTs and the GMT, the design and implementation of the GIPSY instance bootstrap process, the GIPSY node registration process, the GIPSY tier allocation and deallocation processes.

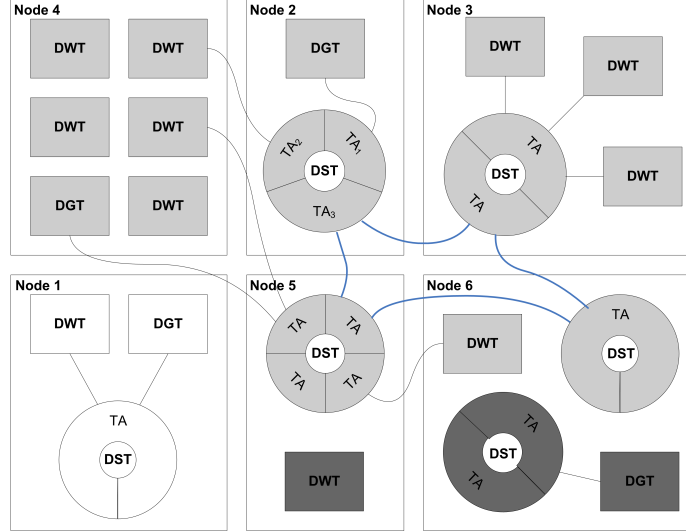


Figure 5: Example of GIPSY instances reproduced from [1]

1.3 Scalability Evaluation

Scalability is an important attribute of a computing system as it represents the ability to achieve long-term success when the system is facing growing demand. The multi-tier architecture was adopted for the GIPSY runtime system for research goals such as scalability [1, 34], therefore once implemented, the scalability of the GIPSY runtime system needs to be assessed for the validation of the implementation. The scalability of the GIPSY runtime system shall be clearly defined; otherwise the assessment of scalability is left to intuition. Besides, in order to assess scalability, an appropriate set of metrics shall also be identified to aid our judgment. Therefore several scalability definitions and metrics used in scientific literature are investigated for this purpose in this section.

1.3.1 Definition of Scalability

Although the term *scalability* is widely used and treated as an important attribute in computing community, there is no commonly accepted definition of scalability available [36]. Without a standardized definition, researchers use *scalability* to denote the capability for the long-term success of a system in different aspects, such as the ability of a system to hold increasing amount of data, to handle increasing workload

gracefully, and/or to be enlarged easily [37].

Moreover, in recent years, the methods of scaling a computing system have been classified into two brief categories, *scale-up* and *scale-out* [38, 39, 40]. *Scale-up* refers to upgrading computers such as adding more processors to handle increased workload, whereas *scale-out* usually refers to expanding the system by adding more low-cost and interconnected computers to handle increased workload. Such classification is useful in discussing the scalability definitions shown below.

Many attempts have been made to define scalability ever since the question “What is scalability” was proposed by Mark D. Hill [41] back in 1990, some of which focused on a generic scalability concept for computer and/or software systems. For example, Bondi described four general types of scalability covering different aspects of a system [37] as shown below.

- Load scalability: the ability of a system to achieve desired performance under increasing loads while making good use of available resources. The *load* here indicates the traffic or workload caused by the inputs of the system.
- Space scalability: the ability of a system to support increasing amount of data without its memory requirements growing into intolerable levels.
- Space-time scalability: the ability of a system to maintain satisfactory performance while the number of objects that it encloses increases by orders of magnitude, e.g. a search engine based on linear search is not space-time scalable, but that based on hash table may be space-time scalable.
- Structural scalability: the ability of a system to expand in a chosen dimension, i.e. increasing the number of components it encompasses, at least within a chosen time frame and without major modifications to its architecture.

Such separation of scalability types is reasonable because a complex system, for example a distributed system with multiple users, servers and databases, naturally has limits in different aspects that are crucial to the system’s long term success, therefore a system may simultaneously possess all of these types of scalability. It

was argued in the paper [37] that the load scalability and the space scalability have no causal relationship between each other, whereas poor space-time scalability or poor structural scalability may result in poor load scalability, but good space-time scalability or good structural scalability does not guarantee good load scalability. Although the paper did not provide an effective methodology to identify scalability metrics, its four types of scalability, especially the load scalability, covered most of the understanding of scalability as shown in the following examples, and thus are very helpful in defining the scalability of the GIPSY runtime system.

Another attempt to define a generic scalability for software systems was made by Duboc *et al.* [36]. They proposed a general framework to characterize and analyze scalability. For a particular software system, their framework sorted the measurable characteristics of this system into predefined concept groups including:

1. scaling dimensions: scaling aspects of the system such as the number of concurrent users or nodes.
2. nuisance variables: characteristics of the system that cannot be changed, such as processor speed.
3. dependent variables: conventional software metrics such as performance, memory consumption that are affected by changes in scaling dimensions.

With the sorted characteristics, scalability of the system was assessed by the analysis of dependent variables with respect to the variation in scaling dimensions, based on which a claim regarding scalability could be made as:

A certain system is scalable with respect to some dependent variables (performance metrics such as response time/query) because it can maintain these dependent variables at certain level while increasing some scaling dimensions (e.g. number of queries).

The framework also used a weighted sum to combine dependent variables of different aspects together and normalized the sum to denote an overall scalability.

For example, in Duboc’s paper, two implementations of the same banking system were compared, and by measuring normalized compound scalability combining the performance and memory consumption over the increasing number of data entities, i.e. in this example the overall scalability was the combination of the load scalability and the space scalability. Therefore, although this framework provided a systematic way to separate scalability-relevant concerns into concept groups to discover their causal relations, their way of combining different types of scalability into a single overall scalability is not suitable for the GIPSY runtime system at the current stage because the weights of the sum are subjective and may be changed from researcher to researcher, and it is better to leave different types of scalability as what they are to allow more flexibility in the assessment of scalability.

In a third attempt, a mathematical representation of scalability was proposed defining scalability as the direction derivative along a particular dimension in a multi-dimensional space [42]. It sorted all the factors of a system into 2 vectors, *problem space* that generalizes the *problem size* such as complexity of an algorithm, and *system space* that generalizes the *system size* such as the number of processors. Based on these two vectors, a function was used to denote the performance of a particular algorithm with particular computer architecture, and the scalability was defined as the derivative of this function. This mathematical representation of scalability gave a useful hint of how to mathematically represent a generic scalability, but essentially did not add new aspects to scalability definitions and therefore is not adopted for defining scalability here.

Based on the above descriptions and discussions of the scalability definitions, we adopt the four types of scalability described in the Bondi’s manuscript [37] to the GIPSY runtime system, because they covered important aspects of the system such as performance, memory consumption, number of nodes and together they gave a more comprehensive assessment of the scalability than other definitions. Still, these four types of scalability need to be adapted for the GIPSY runtime system and software metrics need to be identified so that the four types of scalability can be clearly defined and assessed, therefore several scalability metrics and evaluation methodologies are

studied in the next section.

1.3.2 Scalability Metrics

Scalability metrics are necessary for evaluating scalability because they provide a quantitative basis to reduce subjectivity in the assessment of scalability.

For **space scalability**, as described in Section 1.2.2, in the GIPSY runtime system, all the *computed* demands are stored in the DSTs for future reuse, each DGT has a Local GEER Pool to store GEERs for processing intensional demands, each DWT has a Procedural Class Pool to store definitions of procedures for processing procedural demands, and both the DGT and the DWT have a Local Demand Store to cache outgoing demands. Therefore there are concerns such as how the memory requirement of the GIPSY runtime system changes with the increasing amount of demands stored in the DSTs and in the Local Demand Stores of the DGTs and the DWTs. However, since the GIPSY runtime system is demand-driven and all the demands are stored in and migrated via DSTs, the space scalability of DSTs regarding how their memory requirement changes with increasing demand storage is of top priority, and thus is investigated in this thesis; other space-scalability relevant aspects shall be investigated in future work. The metric for space scalability is therefore the ratio between *total demand size* and *total DST memory size*, where the *total demand size* indicates the overall size of the demands stored in a DST, and the *total DST memory size* indicates the amount of memory available to the DST, e.g. its heap memory size. This ratio can easily indicate which type of DSTs offers a better space scalability for the GIPSY runtime system. Also, the *maximum demand size* of a single demand that a DST with limited memory can store is also of our interest, therefore the *maximum demand size* is the metric for this aspect of the space scalability.

For **space-time scalability**, as the amount of demands stored in a DST increases, if the demand response time, i.e. the demand sending time plus the result reading time, does not increase accordingly, then the DST is considered space-time scalable because its performance is not negatively affected by the amount of objects it stores;

otherwise it is not space-time scalable. The metrics for this type of scalability are therefore *demand response time*, which is the sum of *demand sending time* and *result reading time*.

For **structural scalability**, since the GIPSY runtime system can be enlarged by allocating more tiers and tiers including DGTs and DWTs are interconnected by the DSTs, the maximum number of concurrently connected DWTs and/or DGTs that a DST can support as well as the maximum number of node/tier registrations that a typical GMT can store are used to estimate the maximum size of the system. Therefore the metrics for this type of scalability are *maximum node/tier registrations* and *maximum DST connections*.

As for **load scalability**, it was discovered that there are many publications discussing this type of scalability, and most of them assessed load scalability via the relation between the *throughput* and the amount of computational resources such as the number of processors or software components. For example, based on the limits in existing scalability metrics, a further generalized scalability metric was proposed focusing on productivity [43]. The productivity was represented as the ratio between the optimized performance and the system cost, where the performance was based on the *throughput* such as response per second, and the system cost is usually proportional to the amount of computational resources such as the number of processors.

Similarly, a scalability metric combining both algorithm and hardware system was proposed [44] based on the assumption that for a scalable system, its communication overhead should not increase faster than the increase in the computational, i.e. productive, workload. For a particular system and for a particular algorithm, the metric investigates the relation between the computed speed of a workload and the overall processor speed of the system, where the computed speed is the ratio between the workload and the corresponding execution time. Therefore if the computed speed is regarded as the *throughput*, this metric measures the load scalability via *throughput* and the amount of computational resources measured by the total processor speed.

Besides the above two papers, the relation between the *throughput* and the amount of computational resources in the sense of software components performing the computation was also used in other publications to measure the load scalability. For example:

1. To compare the load scalability of 5 different Enterprise JavaBeans implementations of an e-commerce site, for each implementation, the relation between the *throughput* in *requests/minute* and the number of clients was studied [45]. Since the session beans used to implement the e-commerce site was created upon each client connection, the number of clients implied the amount of the server components, i.e. the beans, running inside the server host.
2. To compare the scalability between the scale-up and the scale-out solutions for a search engine, a single driver generating fixed number of outstanding queries was used, and the relation between the *throughput* in *queries/second* and the number of backends was studied [39].
3. Similarly, a number of concurrent clients were used to test an e-commerce application in scale-out environments of different architectural configurations, with each client periodically sending request and receiving response [40]. The scalability was assessed via the relation between the *throughput* measured in *pages/second* and the number of server nodes.

All the above load scalability evaluation methodologies show that the relation between the *throughput* and the amount of computational resources is an important performance metric in evaluating load scalability. Since in the GIPSY runtime system, demands can be generated in one tier and are processed in other tiers, the tiers that process the demands are the software components that perform the computations. For example, in the case of a particular type of procedural demands, after the *pending* demands are generated by the DGT, these demands are sent into the DST to be picked up and processed by the remote DWTs, after which the *computed* results are put back into the DST and are received by the DGT. Therefore in this case, the number of

DWTs connecting to the DST can be used to indicate the amount of computational resources, and how many corresponding *computed* results that the DGT can receive within a certain time period can be used as the *throughput*. The load scalability of the GIPSY runtime system is assessed by investigating the relation between the *throughput* and the *number of DWTs*.

1.3.3 Scalability Evaluation of the GIPSY Runtime System

Based on the scalability definitions and metrics investigated and adopted for the GIPSY runtime system, the four types of scalability of the GIPSY runtime system and their metrics are summed up below. Detailed definitions, metrics, tests and discussions are presented in Chapter 3.

The metrics defined so far for the four types of scalability are:

- *total demand size*: the size of all the demands stored in a DST. If all the demands stored in the DST are of the same demand size denoted as the *single demand size*, and the amount of demands is denoted as the *number of demands*, then *total demand size* = *number of demands* × *single demand size*. The value of this metric begins from zero, and its unit is a standard byte multiple such as mega bytes (MB).
- *total DST memory size*: the size of memory available to a DST. If the DST is a Java Virtual Machine (JVM) process, then the *total DST memory size* equals the heap memory size of the JVM process. The value of this metric begins from zero, and its unit is a byte multiple such as mega bytes (MB).
- *maximum demand size*: the maximum size of a single demand that can be stored in the DST. If the DST is a JVM process, then the *maximum demand size* is determined by the largest heap generation size as discussed in Section 3.2. The value of this metric begins from zero, and its unit is a standard byte multiple such as mega bytes (MB).

- *demand sending time*: the time spent in sending a demand to a DST. If the Transport Agent of the DST provides a method `setDemand()` to send a demand to the DST, then the time interval from the time when this method is invoked to the time when the method returns is regarded as the *demand sending time*. The value of this metric begins from zero, and its unit is millisecond.
- *result reading time*: the time spent in reading a result from a DST. If the Transport Agent of the DST provides a method `getResult()` to get a result from the DST, then the time interval from the time when this method is invoked to the time when the method returns is regarded as the *result reading time*. The value of this metric begins from zero, and its unit is millisecond.
- *demand response time*: the *demand sending time* plus the *result reading time*. This metric indicates how long the DGT that generates the *pending* demand has to wait until it gets the corresponding *computed* result. The value of this metric begins from zero, and its unit is millisecond.
- *maximum node/tier registrations*: the maximum number of node/tier registrations that a GMT can support. The value of this metric begins from zero.
- *maximum DST connections*: the maximum number of connections that a DST can support. Since DGTs and DWTs can be connected to the DST, this number indicates the maximum number of DGTs/DWTs that the DST can support. The value of this metric begins from zero.
- *throughput and number of DWTs*: given a particular type of procedural demands, after a DGT generates these *pending* demands, the number of corresponding *computed* results that the DGT can receive within a certain time period is denoted as the *throughput* of the demand computation. Also, the number of DWTs that connect to the DST to receive and process the procedural demands is denoted as the *number of DWTs*. The value of the *throughput*

begins from 0 and its unit is *demand/second*, and the value of the *number of DWTs* begins from 0.

Given the metrics defined above, the four types of scalability are:

- Space scalability: the ability of GIPSY runtime system to store increasing amount of demands with tolerable memory usage. The judgment of “tolerable” is described in Section 3.1. The metrics to assess this kind of scalability are *total demand size* and *maximum demand size* versus *total DST memory size*.
- Space-time scalability: the ability of GIPSY runtime system to maintain its performance while the number of demands stored increases. The metrics used to assess this type of scalability are *demand response time* consisting of *demand sending time* and *result reading time*.
- Structural scalability: the ability of GIPSY runtime system to allocate more GIPSY tiers over more GIPSY nodes. The *maximum node/tier registrations* as well as the *maximum DST connections* are the metrics for this type of scalability.
- Load scalability: the ability of GIPSY runtime system to achieve a desired demand processing throughput that is able to increase proportionally with the number of the software components that process the demands. The metrics used to assess this type of scalability are the *throughput* and the *number of DWTs* described above.

1.4 Thesis Overview

1.4.1 Scope

This thesis covers the implementation of the multi-tier architecture of the GIPSY runtime system and the assessment of the scalability of the implemented GIPSY runtime system.

1.4.2 Problem Statement

The proposal and adoption of the multi-tier architecture for the GIPSY runtime system aims at research goals such as scalability. To achieve the research goals, the multi-tier architecture must be implemented. However, before the work presented in this thesis, several problems remained in the GIPSY runtime system.

The first problem was the lack of a deployable GIPSY instance. As stated in Section 1.2.2 a GIPSY instance is a GIPSY runtime system consisting of multiple GIPSY tiers allocated in multiple GIPSY nodes (computers), and a “deployable” GIPSY instance enables the user to register computers as new GIPSY nodes, to allocate and deallocate GIPSY tiers at runtime. Although some classes and API for the multi-tier architecture were defined as stated in Section 1.2.2, the implementation of the new architecture was incomplete before the work presented in this thesis: DSTs based on the previous Jini and JMS DMSs (see Section 1.2.1) were not implemented and integrated into the multi-tier architecture, and the GMT was also not implemented to perform managerial tasks, needless to say the GIPSY bootstrap process, GIPSY node registration, GIPSY tier allocation and deallocation that require an operational GMT. All these tiers and managerial tasks are essential to deliver a deployable GIPSY instance that is able to expand easily among computers across a network.

Besides, to adapt the previously implemented Jini DMS and the JMS DMS (see Section 1.2.1) into the multitier architecture, the incompatibility between the Jini DMS and the JMS DMS must be resolved: the TAs connecting to their Demand Spaces did not inherit the same generic TA interface; the *demand* they migrated did not inherit the same generic *demand* interface as well. Such incompatibility impeded implementation transparency required by the GIPSY runtime system. Also, the Demand Dispatcher in the Jini DMS was invoked by the TA to connect to the Demand Space, whereas in the new multi-tier architecture the Demand Dispatcher is supposed to be a component of the DGT to invoke the TA to connect to the DST, therefore their roles must be switched with each other. Moreover, the Demand

Spaces of the two DMSs, i.e. the JavaSpace service and the JMS broker service were configured and started manually, and whenever there is a critical parameter change such as change in service name and address, the source code of the TAs corresponding to the Demand Spaces must be updated and recompiled as these parameters were hard-coded in the source code of the TAs. Such inflexibility should also be resolved in the multitier architecture.

The goal of this thesis is to solve the problems stated above in order to develop a scalable GIPSY runtime system using the multi-tier architecture, and to assess its scalability to show that the GIPSY runtime system is indeed scalable.

1.4.3 Contributions

The Contributions of this thesis include two parts: the development of a deployable GIPSY runtime system, and the assessment of the scalability of the GIPSY runtime system.

The contributions regarding the development of the GIPSY runtime system include:

- Development of a configuration system for the GIPSY runtime to achieve flexibility in the sense of adding new tier implementations without changing the source code of existing system components.
- Design and implementation of the GIPSY instance bootstrap process to enable the start of the first GIPSY node where the GMT is allocated.
- Design and implementation of the GIPSY node registration, GIPSY tier allocation and deallocation processes to enable the expansion of the GIPSY runtime system over multiple networked computers.
- Implementation of the Jini DST and the JMS DST based on the refactoring of the Jini DMS and the JMS DMS to adapt them to the multitier architecture.

The contributions regarding the assessment of the scalability of the GIPSY runtime system include:

- Study of scalability evaluation methodologies to define the four types of scalability and their metrics applicable to the GIPSY runtime system.
- Assessment of the four types of scalability defined for the GIPSY runtime system via the various tests for each kind of the four types of scalability.

1.4.4 Outline

This thesis consists of five chapters. Specifically:

Chapter 1 introduces the necessary background and the previous work regarding the GIPSY runtime system, as well as the scalability assessment methodologies, and presents the scope, problem statement, contributions and outline of this thesis.

Chapter 2 firstly introduces the tasks required for the development of a deployable GIPSY instance, and then describes the work done to accomplish these tasks, and thirdly shows the tests of the implemented GIPSY runtime system.

Chapter 3 presents the assessment of the four types of scalability of the GIPSY runtime system via several tests, with the test results, and discussions and conclusions.

Chapter 4 shows the conclusions and future work.

Chapter 2

Development of the GIPSY Runtime System

This chapter describes the work of developing the GIPSY runtime system using the multi-tier architecture introduced in Section 1.2.2, so that the scalability of the GIPSY runtime system can be assessed after the development. This chapter first introduces the tasks required to implement the multi-tier architecture, then describes the work done to accomplish these tasks, and finally present the tests to validate the work.

2.1 Requirement Analysis

The purpose of the work described in this chapter is for delivering a deployable GIPSY runtime system using the multi-tier architecture. The multi-tier architecture, the concepts of the four kinds of GIPSY tiers including the General Manager Tier (GMT), the Demand Generator Tier (DGT), the Demand Store Tier (DWT) and the Demand Worker Tier (DWT) as well as the concepts of the GIPSY node and the GIPSY instance were introduced in Section 1.2.2. A scenario of a deployable GIPSY runtime system is that within a Local Area Network (LAN), computers can be registered into the GMT as GIPSY nodes, and once the GIPSY nodes are registered, via the GMT the user is able allocate DGT instances, DWT instances and DST instances in any registered GIPSY nodes at runtime, and is also able to deallocate

the GIPSY tiers allocated in the GIPSY nodes. In this way the GIPSY runtime system can be easily expanded among multiple computers across the network to deal with increasing workload.

The packages and APIs for the multi-tier architecture of the GIPSY runtime system are shown Figure 6. These packages were initially proposed by Han [35] who also developed the prototypes of the DGT and the DWT. The multi-tier packages were further developed by the work presented in this chapter with the design and implementation of the DSTs, the GMT and the GIPSY node to deliver a deployable GIPSY runtime system.

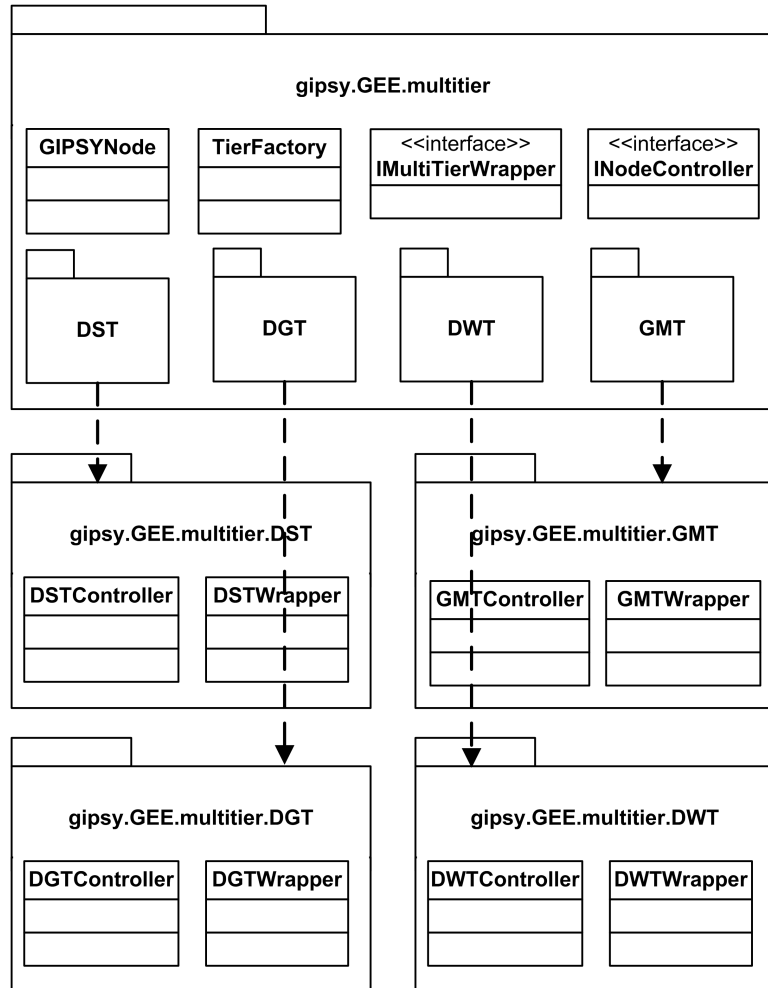


Figure 6: The multi-tier packages of the GIPSY runtime system

The relations among the classes in the multi-tier packages are shown

in Figure 7. As shown in this figure, there is a `GIPSYNode` class, an `INodeController` interface inherited and implemented by the four concrete tier controllers in their corresponding sub-packages, i.e. the `DSTController`, the `DGTController`, the `DWTController` and the `GMTController`, and an interface `IMultiTierWrapper` inherited and implemented by the four concrete tier wrappers in their corresponding sub-packages, i.e. the `DSTWrapper`, the `DGTWrapper`, the `DWTWrapper` and the `GMTWrapper`. Each tier wrapper class stands for a tier type, i.e. the `DST`, `DGT`, `DWT` and the `GMT` respectively, and each tier controller class is used by a `GIPSYNode` instance to create the corresponding tier instances within the GIPSY node, to keep a list of the created tier instances and to destroy the created tier instances when necessary. Therefore as shown in Figure 7, each `GIPSYNode` instance has four different GIPSY tier controllers, each one of which keeps a list of corresponding GIPSY tier instances created by the tier controller via the `TierFactory`. Technically to enable the registration of a computer as a GIPSY node, a process where an instance of the `GIPSYNode` runs is started in the computer in advance, and this process is referred as the `GIPSYNode` process; each tier instance is a separate thread launched by the `GIPSYNode` instance inside the `GIPSYNode` process, or a separate process started by the `GIPSYNode` instance in the same computer where the `GIPSYNode` process is running.

The roles of all the four kinds of GIPSY tiers were described in Section 1.2.2. Among all the four kinds of GIPSY tiers, the `GMT` is vital to the deployment and the expansion of the GIPSY runtime system across networked computers due to the managerial tasks required by its role as the General Manager of the entire runtime system:

- The `GMT` handles the GIPSY node registrations: computers can be registered into the `GMT` as GIPSY nodes, as shown in the use case “Register Node” in Figure 8, and at current stage one `GMT` is sufficient to handle multiple GIPSY node registrations.
- The `GMT` handles the GIPSY tier allocation: as illustrated in the use case

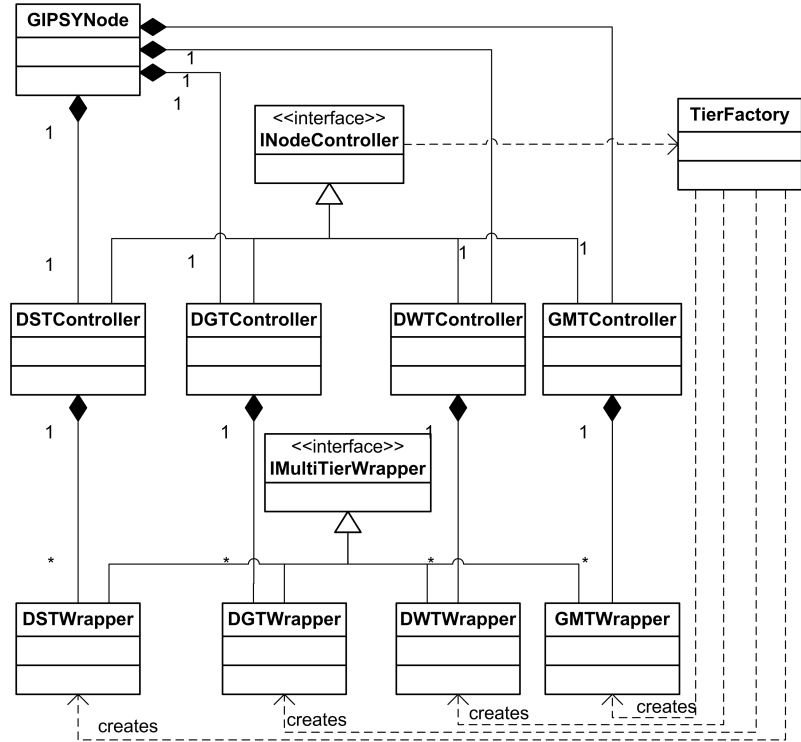


Figure 7: The relations among the classes in the multi-tier packages

“Allocate Tier” in Figure 8, a user can ask the GMT to allocate tiers in a specified GIPSY node and the GMT communicates with the GIPSY node to allocate these tiers. A user interface is required to enable the user interaction.

- The GMT handles the GIPSY tier deallocation: as illustrated in the use case “Deallocate Tier” in Figure 8, a user can ask the GMT to deallocate some tiers that are already running inside a specified GIPSY node, and the GMT communicates with the GIPSY node to deallocate these tiers. Again a user interface is required for the user interaction.

These managerial tasks described in Figure 8 are demand driven due to the design choice made in the proposal of the multi-tier architecture [1]. Specifically, the interactions between the GIPSY node and the GMT in these managerial tasks are carried out via system demands migrated via the DSTs, as introduced in Section 1.2.2. Therefore to implement these managerial tasks, system demands for the GIPSY node registration, tier allocation and deallocation are required, and the DSTs that

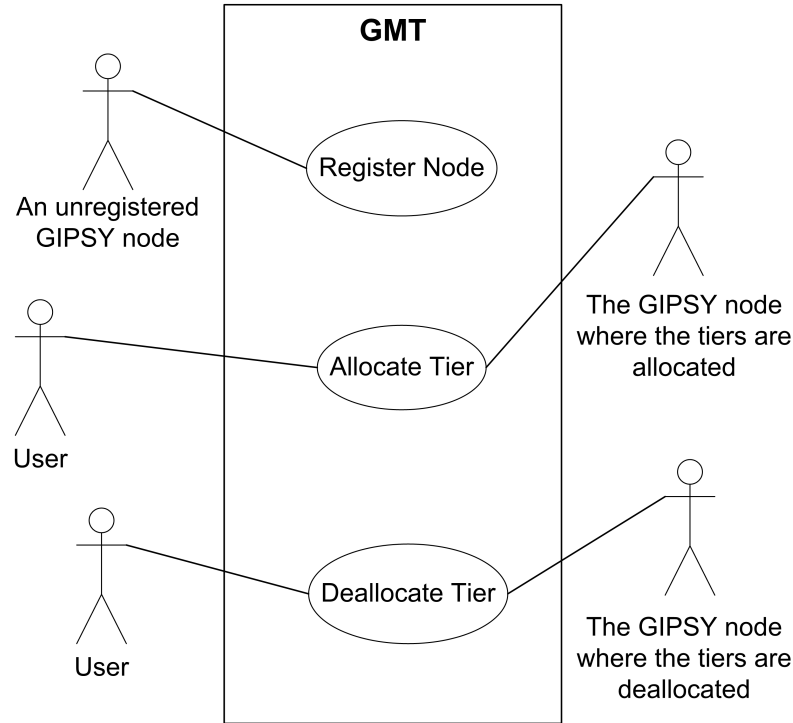


Figure 8: GMT use cases

can be allocated in GIPSY nodes are also required.

However before the work presented in this chapter, the GIPSY tiers of different implementations were integrated with Java Enumeration, which is inflexible to add new tier implementations since the source code of the existing system components affected has to be updated; also, the GIPSY instance bootstrap process, i.e. how a GIPSY runtime system was started from the beginning was not designed; and the node registration, the tier allocation, the tier deallocation, as well as the DST and the GMT were not designed and implemented. Therefore to deliver a deployable GIPSY instance, the following tasks must be performed:

1. Develop a configuration system for the GIPSY runtime system so that new tier implementations can be added into the system without the source-code modification of existing system components.
2. Design the GIPSY instance bootstrap process, i.e. describe how the first GIPSY node where the GMT is allocated is started and registered.

3. Design the GIPSY node registration process. The node registration process is demand driven due to the design choice proposed with the multi-tier architecture [1], therefore the system demands required for the node registration process are also designed in this task.
4. Design the GIPSY tier allocation and deallocation processes. These processes is also be demand driven due to the design choice proposed with the multi-tier architecture [1], therefore the system demands required for the tier allocation and deallocation processes are also designed in this task.
5. Implementation of the DSTs. Since the node registration, tier allocation and deallocation processes are all demand-driven, and demands are stored in and dispatched via DSTs, therefore to implement these managerial processes, the DSTs need to be implemented first. Since the former Jini DMS and the JMS DMS introduced in Section 1.2.1 already have similar concepts such as the Transport Agents (TAs), the two DMSs need to be adapted into the multi-tier architecture.
6. Implementation of the GMT and the GIPSYNode for the GIPSY instance bootstrap process, the node registration, the tier allocation and deallocation processes designed in Task 2, 3 and 4. This task also involves the implementation of the system demands designed in Task 2, 3, 4, and requires that the DTSSs must be implemented before the commence of this task.
7. Test the GIPSY bootstrap process, the GIPSY node registration process, the GIPSY tier allocation and deallocation processes to validate the implementation.

Based on the tasks listed above, the following section describes the work performed to accomplish these tasks.

2.2 Design

2.2.1 Configuration System

To easily add new tier implementation into the GIPSY runtime system, a configuration system based on the Java Reflection [46] and the GIPSY Configuration was developed for the GIPSY runtime system. The GIPSY Configuration class was initially proposed, designed and implemented by Serguei Mokhov based on `java.util.Properties`.

The idea behind the configuration system is that: each tier instance has a corresponding `Configuration` instance that contains a publicly accessible property indicating what tier implementation class can use this `Configuration` instance, and also contains other configuration settings that can only be interpreted by the tier implementation class itself. For example, when a tier `Configuration` instance is received by a `TierFactory`, the `TierFactory` inspects the `Configuration` instance to find which tier implementation class to instantiate, and uses Java Reflection to instantiate the tier implementation with the `Configuration` instance as the argument to the constructor of the tier implementation. Once the tier implementation is instantiated, it looks up in the `Configuration` instance for other configuration settings to perform its own tasks accordingly. In this way new tier implementations can be easily added without changing the source code of other classes such as the `TierFactory` who instantiates these tier implementations.

Also, since `Configuration` instances can be easily written to and read from files, configuration settings can be easily updated to cope with changes by setting corresponding configuration properties such as service names, ports and addresses, either manually in a configuration file or automatically by a program, without changing the source code of tier implementations who use these configuration settings. The configuration files for each tier implementation are therefore created, and the paths of these configuration files are also specified in the configuration file for the `GIPSYNode` as its default tier configurations using which tiers can be instantiated and allocated.

The idea of the tier configuration and the tier configuration files also applies to other classes such as the Transport Agent (TA) implementations. Given a TA configuration, the `TAFactory` can instantiate a TA instance without knowing the actual TA implementation at compile time, therefore new TAs can be easily added into the GIPSY runtime system without source code modification for the `TAFactory`.

Figure 9 shows the idea of the configuration files: *DST1.config* and *DST2.config* are configuration files for two different DST implementations, and the *DST1TA.config* and the *DST2TA.config* are the configuration files of their corresponding TAs; the only one *GMT.config* in the figure is to indicate the scenario when there is only one GMT implementation available to the `GIPSYNode`.

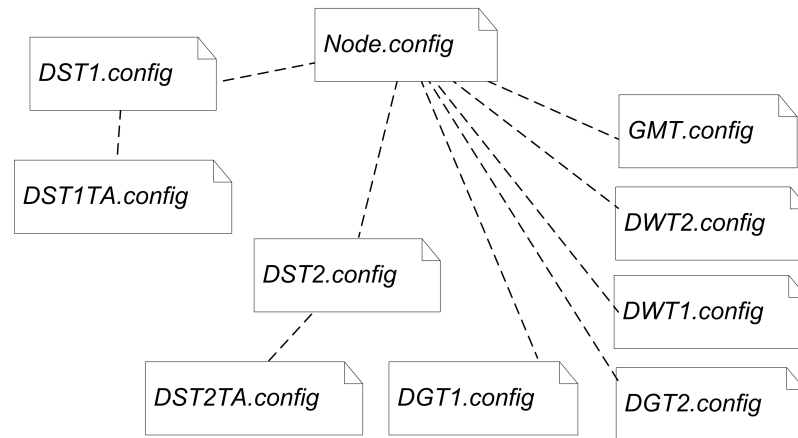


Figure 9: Configuration files of the GIPSY runtime system

Inside each configuration file, the names of the configuration properties (or settings) are arranged in a hierarchical structure similar to how the source code packages were arranged to ease the addition of new properties. For example, Figure 10 shows the hierarchy of some configuration properties of the JMS DST introduced in Section 2.3.2: the configuration properties were defined hierarchically. However, since configuration properties are stored in a flat `java.util.Properties` instance, such hierarchical arrangement is for ease of addition only rather than for ease of search.

Equipped with the flexible configuration system, further design of the GIPSY runtime system is explained in the following sections.

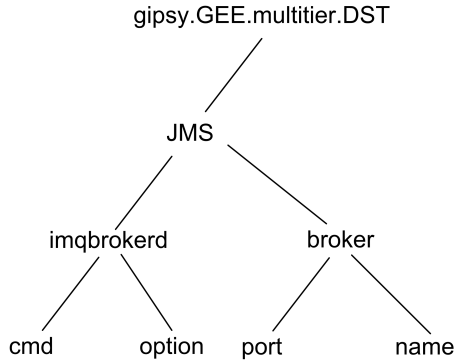


Figure 10: Configuration property naming structure

2.2.2 GIPSY Instance Bootstrap Process

As stated in Section 1.2.2 a GIPSY instance is a GIPSY runtime system deployed in multiple registered computers (called GIPSY nodes) with multiple GIPSY tiers allocated in these nodes that together can execute GIPSY programs. Since each GIPSY node is a computer running a GIPSYNode process that interacts with the remote or local GMT, once the GIPSYNode process is started, it is registered to the GMT to enable further communication. However, since the GMT is also allocated within a GIPSY node, a dilemma similar to the “chicken or the egg” arises when considering how the GIPSY instance is bootstrapped, i.e. how the first GIPSYNode process is started and registered.

The design of the bootstrap process presented in this section assumes that the first GIPSYNode process does not need to be registered when it is started. Instead, the user can ask this GIPSYNode process to allocate a GMT who will then automatically register this GIPSYNode process. The flowchart of the bootstrap process is shown in Figure 11.

In the bootstrap process illustrated in Figure 11, once the GIPSYNode process is started, it initializes itself by loading all the tier configurations, and if it has a GMT configuration, it prompts the user to choose if to register the node or to start up a GMT. If the user chooses to register this node, how the GIPSYNode process is registered is explained in Section 2.2.3. If the user chooses to start up a GMT, a DST for receiving other node registrations is started using the DST configuration

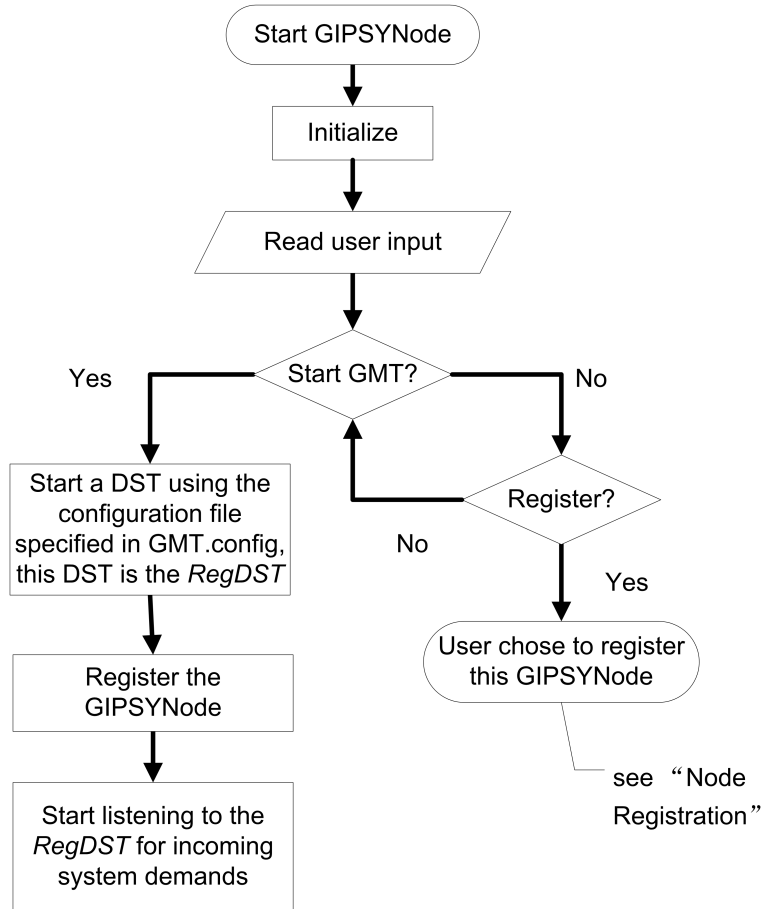


Figure 11: GIPSYNode bootstrap process

file specified in the GMT configuration, and the TA configuration corresponding to the DST is created and written into a configuration file to allow other GIPSY nodes to connect to the DST for the node registration purpose; the GIPSYNode where this GMT is allocated is also automatically registered into the GMT to complete the bootstrap process.

2.2.3 GIPSY Node Registration

As a demand-driven system, the GIPSY node registration process is also demand driven. For each GIPSY instance, there is one GMT and one corresponding DST started in the GIPSY instance bootstrap process described in Section 2.2.2. The DST started in the bootstrap process is used by the GMT to receive system demands

issued by other GIPSY nodes for the node registrations, therefore this DST is referred as the *RegDST* (short for *Registration DST*). For a GIPSYNode other than the one who started the GMT, the TA configuration of the *RegDST* must be available to the GIPSYNode before the node registration so that once started, the GIPSYNode process can instantiate a TA to connect to the *RegDST* to send and receive node registration relevant system demands.

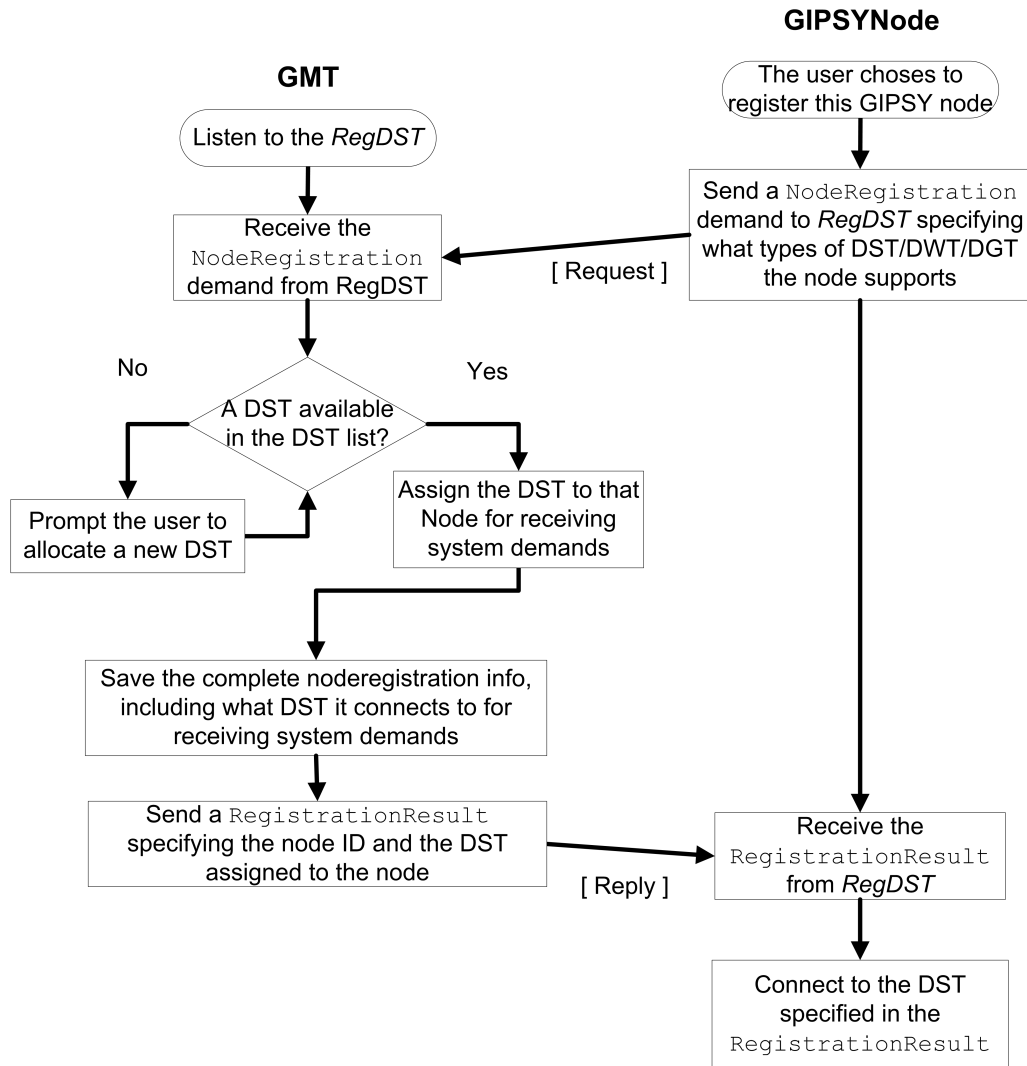


Figure 12: GIPSYNode registration process

As shown in the flowchart of the GIPSYNode registration process in Figure 12, the GIPSY node registration is done via a pair of system demands called *NodeRegistration* and *RegistrationResult*, with both demands sharing the

same demand signature but the state of the former is *pending*, and that of the latter is *computed*. In this process, once the user chooses to register this GIPSY node, the GIPSYNode process sends the system demand `NodeRegistration` as a request to the GMT. Upon receiving the system demand, the GMT assigns an available DST to the GIPSYNode; the GMT then saves the node registration information and replies the GIPSYNode with a `RegistrationResult` demand containing the TA configuration exported by the DST assigned as well as the node ID assigned by the GMT. Upon receiving the `RegistrationResult`, the GIPSYNode connects to the DST assigned by the GMT to listen to other incoming system demands issued by the GMT in the situation of tier allocation and deallocation, as described in Section 2.2.4.

The reason why the GMT assigns a DST to the GIPSY node for sending future system demands rather than always communicating with the GIPSY node via the *RegDST* is that: in order to receive system demands issued from the GMT for purposes such as tier allocation and deallocation, the GIPSYNode must always connect to the assigned DST for receiving any incoming system demands issued by the GMT; however, since the maximum connections supported by a DST is limited as investigated in Section 3.4.2, if only the *RegDST* is used for sending and receiving all the system demands, the *RegDST* will eventually reach its maximum connection capacity and refuse new node registrations, preventing the system from growing larger structurally; therefore to resolve this limitation in the structural scalability (see Section 3.1), during the node registration process, a DST other than the *RegDST* can be assigned to the GIPSY node, and if there is no DST available, the GMT prompts the user to allocate a new DST; such user interaction can be enabled via a user interface such as a command-line console, as shown in Section 2.3.3.

2.2.4 GIPSY Tier Allocation and Deallocation

When GIPSY nodes are registered, new tiers can be allocated inside the GIPSY node, and previously allocated tiers can be deallocated. As shown in the flowcharts of the tier allocation process in Figure 13, the tier allocation process is done via a pair of system demands, `TierAllocationRequest` and `TierAllocationResult`,

with both demands sharing the same demand signature but the state of the former is *pending* and the state of the latter is *computed*. The `TierAllocationRequest` demand specifies the node ID of the GIPSY node where the tiers are to be allocated, and also specifies the type of the tier, the tier configuration and how many tier instances are to be allocated, i.e. allowing tier instances of the same type using the same configuration file to be allocated in a batch to save time. The corresponding `TierAllocationResult` demand contains a list of tier registrations, each tier registration containing information such as the tier ID assigned internally by the GIPSY node for each tier instance allocated.

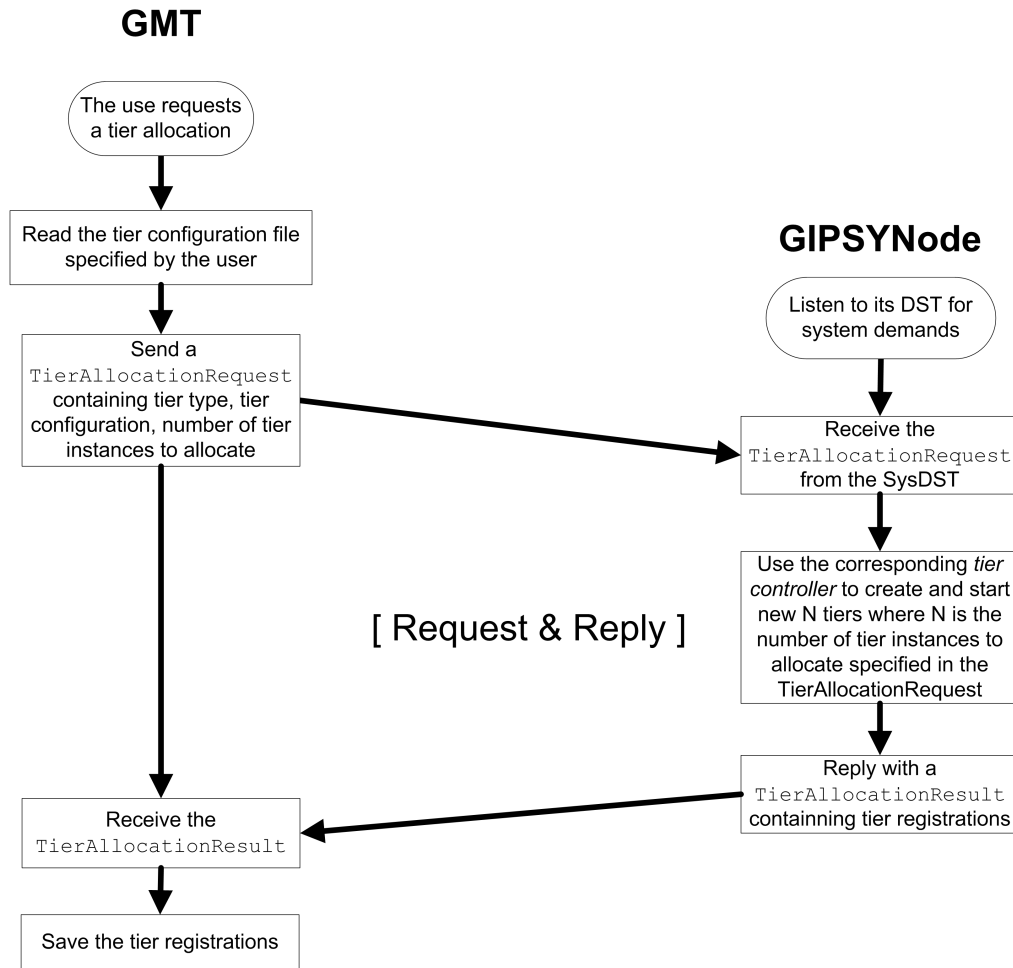


Figure 13: GIPSY tier allocation process

In the tier allocation process described in Figure 13, when the user asks the GMT via a user interface such as a command-line console to allocate new tiers

inside a registered GIPSY node, the GMT issues a `TierAllocationRequest` demand to the DST associated with the GIPSY node. Upon receiving the `TierAllocationRequest` from its associated DST, the `GIPSYNode` process uses the corresponding tier controller, i.e. the `DSTController`, the `DGTController` or the `DWTController`, according to the tier type specified by the `TierAllocationRequest`, to allocate the specified number of tier instances using the specified tier configuration, and replies the GMT with a `TierAllocationResult` demand containing the tier registration information such as the tier IDs assigned by the GIPSY node itself. Upon receiving the `TierAllocationResult`, the GMT saves the tier registrations for future use.

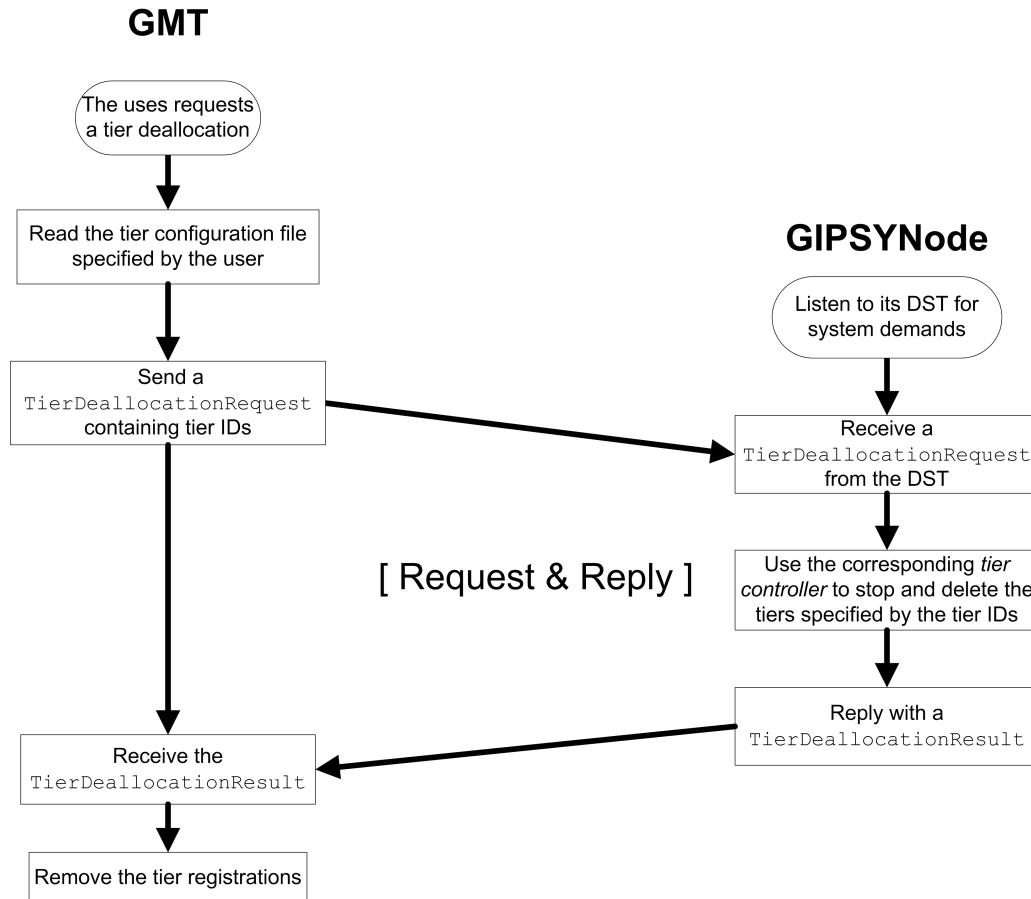


Figure 14: GIPSY tier deallocation process

Similar to the tier allocation process, the tier deallocation process is done via a pair of system demands, `TierDeallocationRequest` and

`TierDeallocationResult`, as shown in the flowchart of the tier deallocation process in Figure 14. In this process, when the user asks the GMT via a user interface such as a command-line console to deallocate the previously allocated tiers from a registered GIPSY node, the GMT issues a system demand `TierDeallocationRequest` to the DST associated with the GIPSY node. The `TierDeallocationRequest` specifies the tier type and the tier IDs of the tier instances to be deallocated. Upon receiving the system demand, the `GIPSYNode` process uses the corresponding tier controller to deallocate the tiers specified, and replies the GMT with the `TierDeallocationResult`. Upon receiving the `TierDeallocationResult`, the GMT removes the registration information of the deallocated tiers.

2.3 Implementation

The implementation of the designs presented in Section 2.2 started from the refactoring of the Jini DMS and the JMS DMS introduced Section 1.2.1, therefore this section firstly introduces the refactoring work of the Jini DMS and the JMS DMS, then the implementation of the Jini DST and the JMS DST, and finally the implementation of the GMT and `GIPSYNode`.

2.3.1 Refactoring the Jini DMS and the JMS DMS

The DSTs are critical to the implementation of a deployable GIPSY runtime system because all the demands are stored in and migrated via the DSTs, and even the GIPSY instance bootstrap process requires the existence of DSTs as explained in Section 2.2.2. Since some of the DST relevant concepts in the multi-tier architecture (see Section 1.2.2) such as the Transport Agent (TA) already existed in the former Jini DMS and the JMS DMS as introduced in Section 1.2.1, these two DMSs needed to be adapted to meet the requirements in the multi-tier architecture.

The initial refactoring of the Jini DMS and the JMS DMS was done in collaboration with Serguei Mokhov and Bin Han. The overall refactoring process

went through several major phases:

1. Resolving the incompatibility between the Jini TA and the JMS TA by making them inherit the same interface, `ITransportAgent`; also resolving the incompatibility between the `JiniDemandDispatcher` and the `JMSDemandDispatcher` by making them inherit the same interface, `IDemandDispatcher`;
2. Refactoring the Jini TA so that it connected directly to its corresponding Demand Space, i.e. the `JavaSpace`; refactoring the `JiniDemandDispatcher` so that it no longer directly connected to the `JavaSpace` directly, instead it connected to the `JavaSpace` via the Jini TA;
3. Generalization of the business logic of the `JiniDemandDispatcher` and the business logic of the `JMSDemandDispatcher` so that any one of them can use any one of the Jini and JMS TAs, i.e. the Demand Dispatcher implementations deal with `ITransportAgent` only and are no longer aware of the heterogeneity of TA implementations.

The refactoring work presented above was just a warm-up of the adapting of the former DMSs into the multi-tier architecture, since it only adapted the TAs and the Demand Dispatchers towards the multi-tier architecture. Further development of deployable DSTs is presented in Section 2.3.2.

2.3.2 Implementation of the Jini and the JMS DSTs

In the former Jini and JMS DMSs introduced in Section 1.2.1, the Demand Spaces, i.e. the `JavaSpace` service and the JMS broker service, were configured and started manually before running the GIPSY runtime system, and whenever there were parameter changes in such as the `JavaSpace` service name and address or the JMS message queue name and address, the corresponding parameters hard-coded in the TAs would have to be updated as well. To resolve such inconvenience and inflexibility, a Jini DST and a JMS DST that can be allocated via a system demand were developed

in the work presented in this section, and the TAs connecting to these DSTs were also designed to be flexible in the sense that they can set up connections at runtime using the TA configurations exported by the DSTs so that they do not need to change their source code to cope with connection-relevant parameter changes.

The JavaSpace service used for the Jini DST is Apache River v2.1.2 [47], and the JMS chosen for the JMS DST is the Sun Java System Message Queue 4.3. Both were chosen due to their ease of deployment and relatively small size. Since the core services, i.e. the JavaSpace service and the JMS broker service are separate processes other than the process where the GIPSYNode is running, the Jini DST and the JMS DST are therefore implemented to be DST wrappers that are able to automatically start and stop these core services via command lines whenever the GMT asks them to do so, and are able to dynamically configure these services as needed.

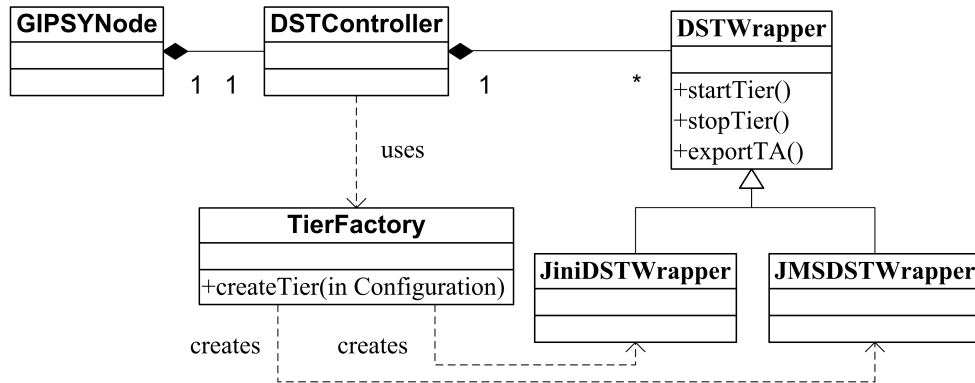


Figure 15: Jini DST wrapper and the JMS DST wrapper

The class diagram of the `JiniDSTWrapper` and the `JMSDSTWrapper` is shown in Figure 15. Each one of the two DST wrappers are instantiated using a `TierConfiguration` instance by the `createTier()` method of the `TierFactory`. When the `startTier()` method of either the DST wrappers instance is called, the DST wrapper first checks if the DST has been allocated or not; if the DST has been allocated, the DST wrapper then checks if the DST is still operational or not; if the DST is not operational, the DST wrapper kills the DST process, otherwise the method returns immediately; if the DST is killed or not allocated, the DST wrapper continues to allocate a new DST by reading and executing the corresponding command lines

contained in the DST configuration. These command lines are preset via configuration files for ease of modification. Each DST core service, i.e. the JavaSpace service or the JMS broker service, is started using a dynamically chosen port, and is assigned a service name that is either automatically generated or preset in the DST configuration. With the dynamic service port and name configuration, multiple DSTs can be easily allocated using the same configuration file. Once the DST is started, the DST wrapper automatically generate the corresponding TA configurations that can be exported via the method `exportTAConfig()`. When the method `stopTier()` is called, the DST wrapper kills the DST process.

The TA configuration exported by the DST wrapper contains connection relevant configuration settings such as the DST service name, host, port, as well as service quality relevant configuration settings such as turning persistence on and off. A DST with the persistence service turned on is referred as a persistent DST, and a DST with the persistent service turned off is referred as a non-persistent or transient DST. Once crashed and restarted, the persistent DST can recover its previous state from its persistent storage; whereas a transient DST will lose its previous state as it saves all the information only in memory. When other GIPSY tiers such as DGT and DWT gets a TA configuration, they use a TA factory to create the corresponding TA instance to connect to the corresponding DST. Figure 16 shows the class diagram of the Jini TA, the JMS TA and the TA factory.

Listing 2.1 shows the contents of the configuration file *JiniDST.config* that is used by the `JiniDSTWrapper` to start a non-persistent JavaSpace service, and the corresponding TA configuration exported by the `JiniDSTWrapper` is shown in Listing 2.2.

The content of *JiniDST.config* shown in Listing 2.1 is based on how the services required by the Jini DST are started: the Jini DST consists of a JavaSpace service and a transaction manager service for reliable demand storage and demand state update, and in order to register and to look up these services, a lookup discovery service is also started. All these Jini services require a HTTP server as the class file codebase for downloading their corresponding *.jar* files, and Apache River v2.1.2 provides such

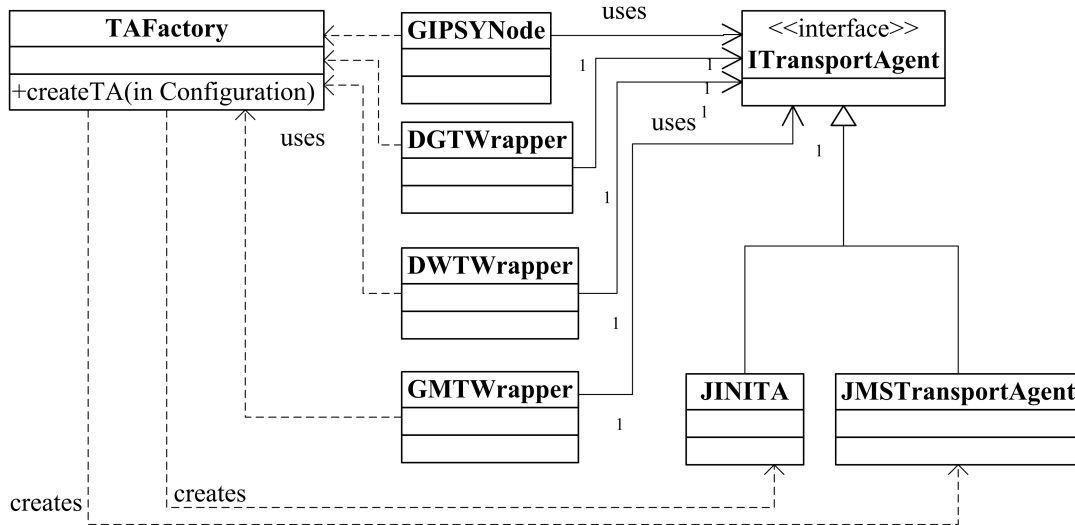


Figure 16: TAs and the TA factory

a HTTP server called the `ClassServer`. To start the `ClassServer` and these services, a Jini configuration file must be passed as the argument to the `start.jar` provided with the Apache River v2.1.2, and this configuration file specifies what services to start and what service implementations to use. The configuration files to start the persistent Jini services and to start the transient Jini services are different: in the `JiniDST.config` shown in Listing 2.2, the `startTransientJini4.config` is used by the `start.jar` to start the transient Jini services, and the `startPersistentJini4.config` is used by the `start.jar` to start the persistent Jini services.

Listing 2.3 shows the contents of the configuration file `JMSDST.config` used by the `JMSDSTWrapper` to start a non-persistent JMS broker service, and the corresponding TA configuration exported is shown in Listing 2.4. Similar to the Jini DST, the user can also specify if the DST persistence is enabled or not. Unlike Jini, the Sun Java Message Queue broker service can be started and configured via command lines, and the complete reference of the instructions and explanations on how to start and configure the broker services is in *Sun Java System Message Queue 4.3 Administration Guide* [48].

With the DSTs and their TAs available, the implementation of the GMT and the `GIPSYNode` is presented in Section 2.3.3.

```

# Set the class that uses this configuration file to be instantiated, so that the
# TierFactory can instantiate the correct DSTWrapper implementation.
gipsy.GEE.multitier.wrapper.impl=gipsy.GEE.multitier.DST.jini.JiniDSTWrapper

# Set the working directory of the DST instance process,
gipsy.GEE.multitier.DST.workingdir=jini/

# Specify the port of the lookup service.
gipsy.GEE.multitier.DST.jini.unicast.port=4162

# Specify the multicast group name of Jini services.
gipsy.GEE.TA.jini.discovery.multicast.group=gipsy

# Set the startup command line without the configuration file passed to Jini's start.
jar
gipsy.GEE.multitier.DST.jini.start.cmd=cmd /c start java -Xms256m -Xmx256m -XX:
NewRatio=1 -XX:SurvivorRatio=1022 -Xss320k -Djava.security.policy=start.policy -
jar lib/start.jar

# To start a persistent Jini DST, pass startPersistent4.config as the value
# of this property.
gipsy.GEE.multitier.DST.jini.start.config=startTransient4.config
#gipsy.GEE.multitier.DST.jini.start.config=startPersistent4.config

gipsy.GEE.TA.jini.isTransactional=true

```

Listing 2.1: Sample JiniDST.config

```

#RegDST TA Configuration
gipsy.GEE.TA.jini.isTransactional=true
gipsy.GEE.TA.jini.discovery.unicast.URI=jini\://HostName\:4162/
gipsy.GEE.TA.jini.discovery.multicast.group=gipsy
gipsy.GEE.TA.jini.discovery.lookup.name= HostName-0
gipsy.GEE.TA.implementation=gipsy.GEE.IDP.DemandGenerator.jini.rmi.JINITA

```

Listing 2.2: Sample content of the configuration of a Jini DST TA

2.3.3 Implementation of the GMT and the GIPSY Node

The implementation of the GMT requires the implementation of the system demands designed in the GIPSY instance bootstrap process, the GIPSY node registration process and the GIPSY tier allocation and deallocation processes as described in Section 2.2. The class diagram of these system demands is shown in Figure 17. The detailed roles of these system demands in the managerial tasks were described in Section 2.2.

As mentioned in the designed managerial processes in Section 2.2, the GMT should be able to save node/tier registrations. As proof of concept, the GMT implemented in this section uses an instance of the GMTInfoKeeper class to store

```

# Set the class that to be instantiated using this configuration file
gipsy.GEE.multitier.wrapper.impl=gipsy.GEE.multitier.DST.jms.JMSDSTWrapper

# Set the working directory of the DST instance process.
gipsy.GEE.multitier.DST.workingdir=jms/

# Specify how to run the command exe used to start or destroy a broker instance
gipsy.GEE.multitier.DST.jms.imqbrokerd.cmd=cmd /c start mq/bin/imqbrokerd

# Specify quality relevant command line options for a broker.
gipsy.GEE.multitier.DST.jms.imqbrokerd.option=-reset store -Dimq.portmapper.backlog
=150 -Dimq.jms.max_threads=10000 -Dimq.message.max_size=-1 -Dimq.autocreate.
destination.maxNumMsgs=-1 -Dimq.autocreate.destination.maxBytesPerMsg=-1 -Dimq.
autocreate.destination.maxBytesPerMsg=-1 -Dimq.autocreate.destination.
maxNumProducers=-1 -Dimq.autocreate.queue.maxNumBackupConsumers=-1 -Dimq.
autocreate.queue.maxNumActiveConsumers=-1 -Dimq.persist.file.sync.enabled=true -
Dimq.persist.file.transaction.memorymappedfile.enabled=false -Dimq.red.threshold
=100 -vmargs "-Xms256m -Xmx256m -XX:NewRatio=1 -XX:SurvivorRatio=1022 -Xss320k"

gipsy.GEE.TA.jms.queue.pending=pending
gipsy.GEE.TA.jms.queue.inprocess=inprocess
gipsy.GEE.TA.jms.queue.computed=computed

gipsy.GEE.TA.jms.isPersistent=false

```

Listing 2.3: Sample JMSDST.config

```

gipsy.GEE.TA.jms.queue.computed=computed
gipsy.GEE.TA.jms.broker.address= HostName\:58283
gipsy.GEE.TA.jms.queue.inprocess=inprocess
gipsy.GEE.TA.jms.queue.pending=pending
gipsy.GEE.TA.implementation=gipsy.GEE.IDP.DemandGenerator.jms.JMSTransportAgent

```

Listing 2.4: Sample content of the configuration of a JMS DST TA

all the node/tier registrations in memory. The class diagram of the GMTWrapper and the GMTInfoKeeper is shown in Figure 18.

To enable user interaction with the GMT, a simple command-line console was also implemented as shown in Figure 18 so that the user can request the GMT to allocate and deallocate GIPSY tiers at runtime. The visual effect of the GMT console is presented in the tests in Section 2.4. The commands that can be entered into the GMTConsole to control the GMT are:

- allocate *NodeID* DST *DSTConfigFile* *HowMany*: this command requests the GMT to allocate *HowMany* number of DST instances in the node specified by the *NodeID* using the DST configuration loaded from the specified *DSTConfigFile*.

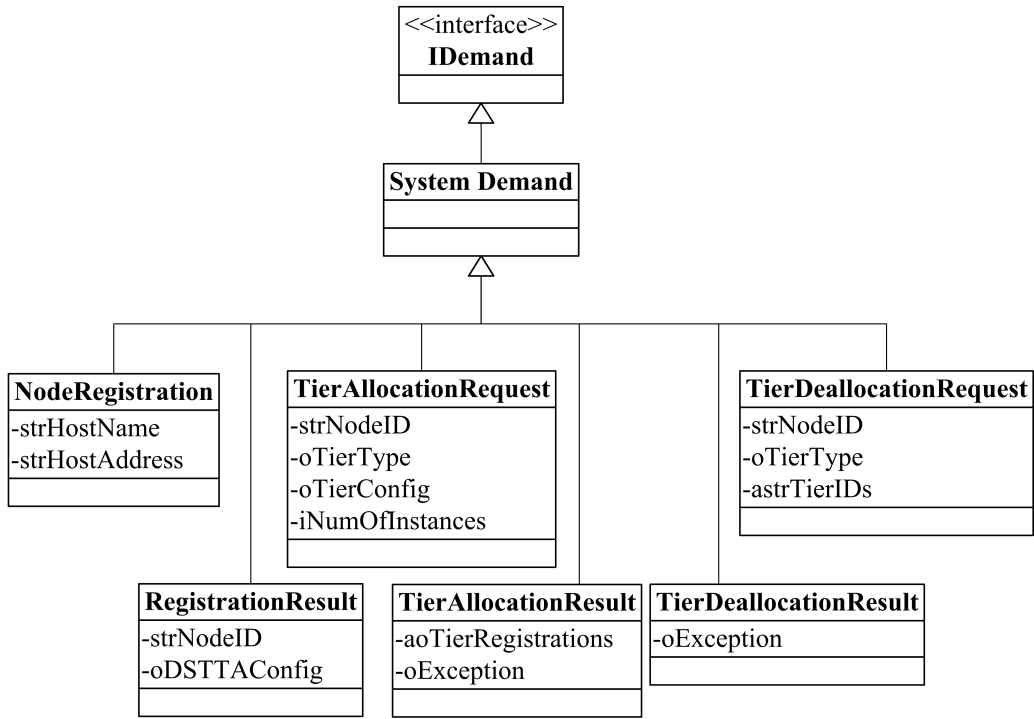


Figure 17: The system demands required by the GMT

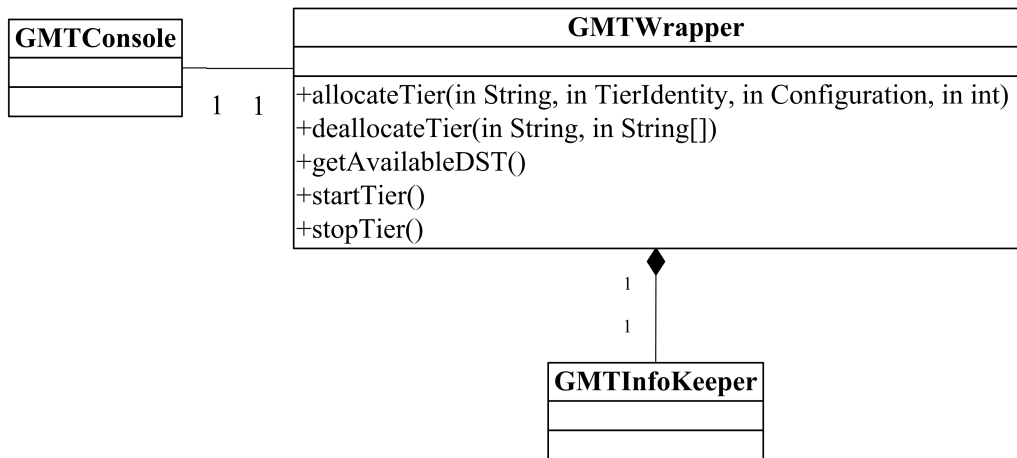


Figure 18: The GMT wrapper class

- `allocate NodeID DWT DWTCfgFile DSTIndex HowMany`: this command asks the GMT to allocate *HowMany* number of DWT instances in the node specified by the *NodeID* using the DWT configuration loaded from the specified *DWTCfgFile*, and all the DWTs allocated connect to the DST specified by the *DSTIndex*.
- `allocate NodeID DGT DGTCfgFile DSTIndex HowMany`: this command asks the GMT to allocate *HowMany* number of DGT instances in the node specified by the *NodeID* using the DGT configuration loaded from the specified *DGTCfgFile*, and all the DGTs allocated connect to the DST specified by the *DSTIndex*.
- `deallocate NodeID TierType TierID1 TierID2 ...`: this command asks the GMT to deallocate the tier instances specified by the *TierID1*, *TierID2*, ... of the type *TierType* in the node specified by the *NodeID*.

The `GIPSYNode` class is instantiated by a `main()` method so that it can be started as a process. It implements all the `GIPSYNode` process relevant business logic designed and described in Section 2.2. When the `GIPSYNode` process is started, its `registerNode()` method can be invoked to register this GIPSY node. Figure 19 shows the sequence diagram of the node registration process.

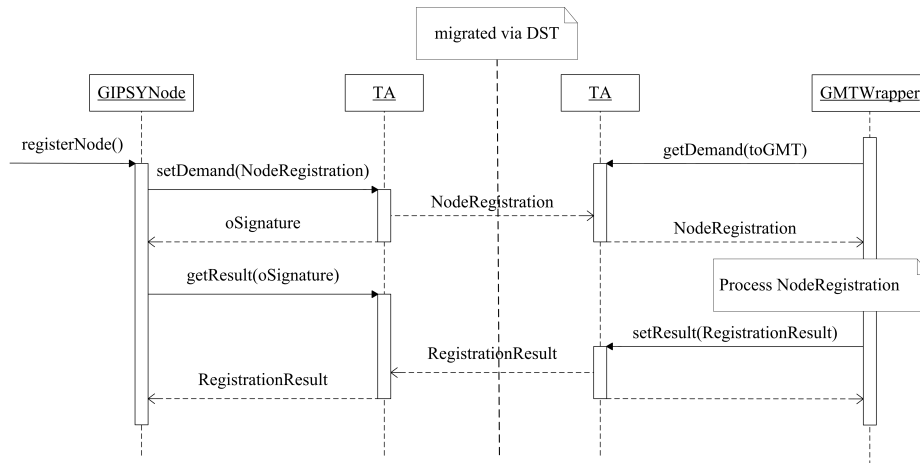


Figure 19: The node-registration sequence diagram

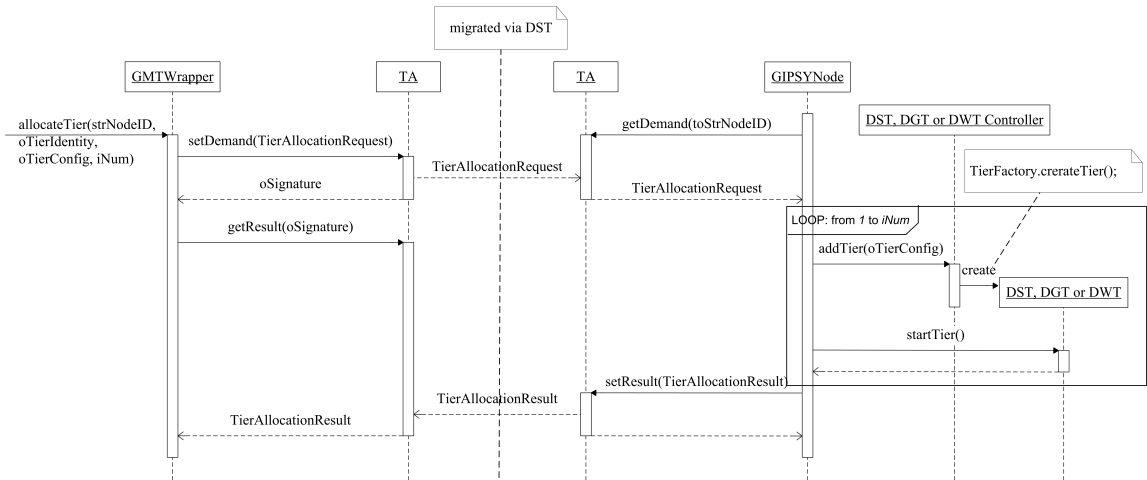


Figure 20: The tier-allocation sequence diagram

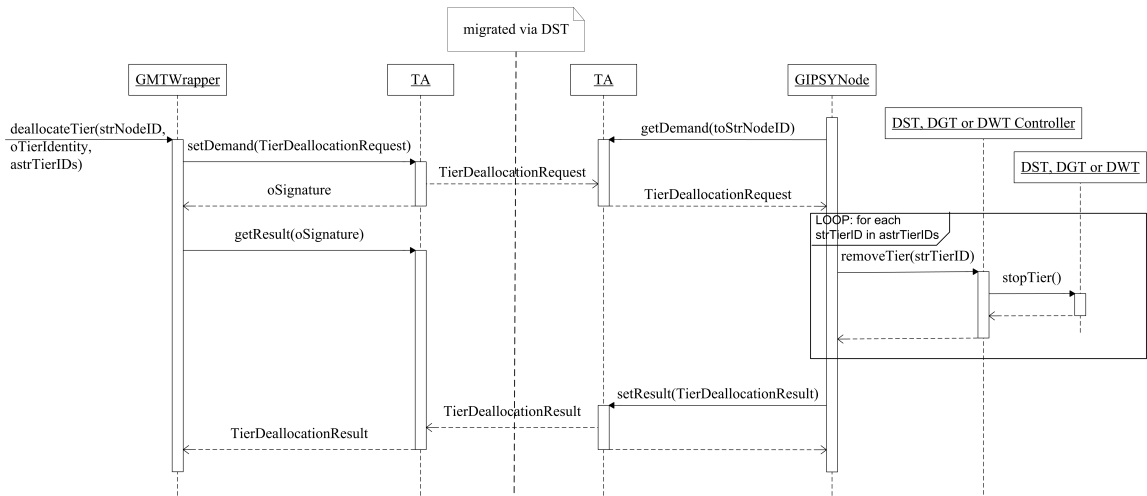


Figure 21: The tier-deallocation sequence diagram

Figure 20 and 21 show the sequence diagrams for the tier allocation process and the tier deallocation process respectively. Whenever the GIPSYNode is asked to allocate a tier by a TierAllocationRequest system demand issued from the GMT, it passes the tier configuration contained in the TierAllocationRequest to the addTier() method of the tier's corresponding tier controller. The tier controller then asks the TierFactory to create a tier instance using the tier configuration via Java Reflection, and starts the tier after the tier is instantiated. For the tier deallocation, the GIPSYNode invokes the removeTier(String) method of the corresponding tier controller that in turn invokes the stopTier() method of the

corresponding tier.

The `GIPSYNode` also support command-line input for the user to control the node interactively. The two most important commands are:

- `start GMT GMT.config`: allocate a GMT using the configuration specified by the configuration file `GMT.config`. This `GIPSYNode` is then automatically registered by the GMT started.
- `register`: register the `GIPSYNode` to a remote GMT.

With the implementation of the GMT and the `GIPSYNode` and all the work presented in Section 2.3, a deployable GIPSY instance is available for the scalability assessment.

2.4 Tests

2.4.1 GIPSY Instance Bootstrap and GIPSY Node Registration

Objective

To test if the GIPSY instance bootstrap process and the GIPSY node registration process is working.

Scenario

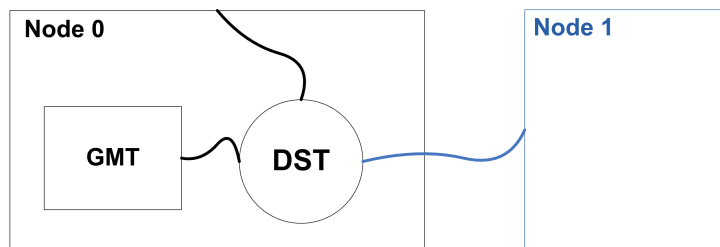


Figure 22: Test scenario for GIPSY instance bootstrap and node registration

As shown in Figure 22, two computers are used as the GIPSY nodes for this test, in which the **Node 0** is used to test the GIPSY instance bootstrap process, and **Node 1** is used to test the GIPSY node registration process. Therefore the test scenario involves two steps:

Step 1: start the GIPSYNode process in **Node 0**, and in the command-line console enter the command:

```
start GMT GMT.config
```

The expected results of this step are: a GMT and a DST are allocated in this bootstrap process in **Node 0**, and the GMT console shows that **Node 0** is registered during the bootstrap process.

Step 2: start the GIPSYNode process in **Node 1**, and in the command-line console enter the command:

```
register
```

The expected results of this step are: the command-line console of **Node 1** outputs that **Node 1** is registered, and the GMT console also outputs that Node 1 is registered.

Environment

The test was performed in two computers with the configuration specified in Table 1

OS Name	Microsoft Windows 7 Enterprise
Version	Version 6.1.7600 Build 7600
System Type	X86-based PC
Processor	Intel(R) Core(TM)2 CPU 6300 @ 1.86GHz, 1862 Mhz, 2 Core(s), 2 Logical Processor(s)
Installed RAM	2.00 GB
Total RAM	2.00 GB
Available RAM	1.06 GB
Total Virtual Memory	4.00 GB
Available Virtual Memory	2.58 GB
Page File Space	2.00 GB

Table 1: Computer hardware and operating system environment

Result

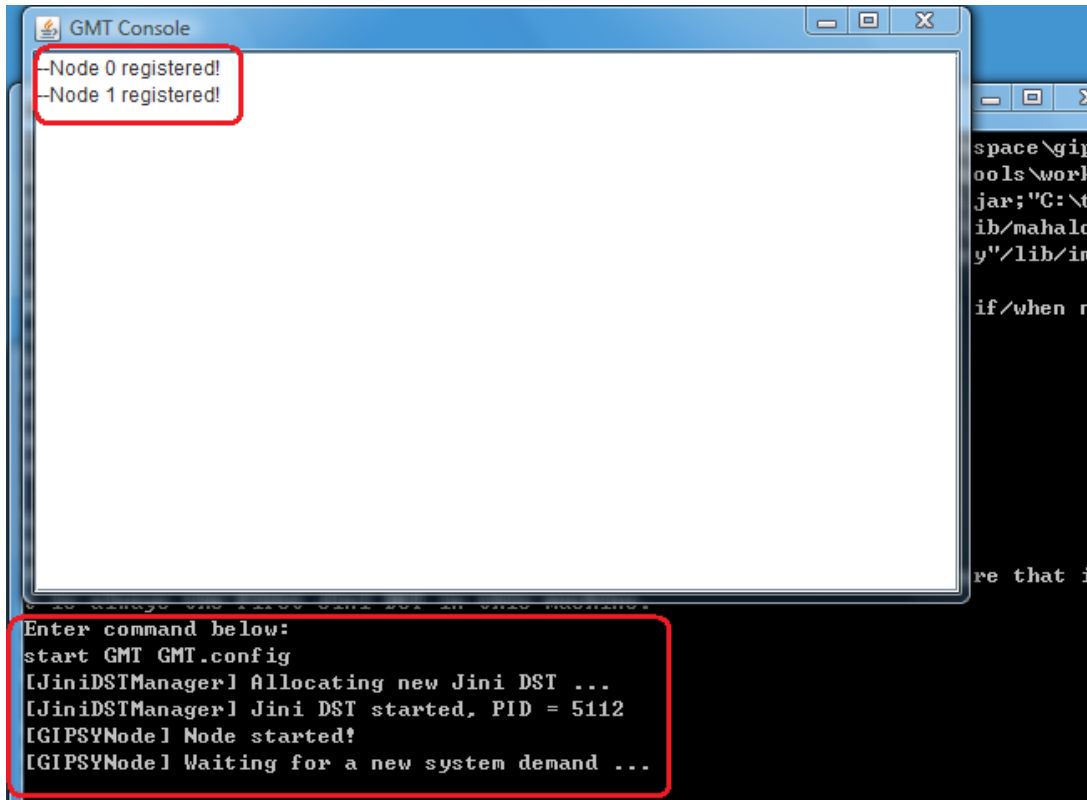


Figure 23(a) displays two console windows. The upper window, titled "GMT Console", shows the following output:

```
-Node 0 registered!  
-Node 1 registered!
```

The lower window shows the command prompt output for starting GMT:

```
Enter command below:  
start GMT GMT.config  
[JiniDSTManager] Allocating new Jini DST ...  
[JiniDSTManager] Jini DST started, PID = 5112  
[GIPSYNode] Node started!  
[GIPSYNode] Waiting for a new system demand ...
```

(a)

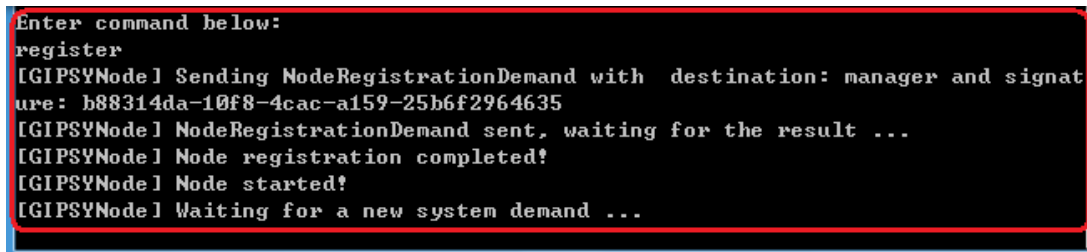


Figure 23(b) displays the console output for Node 1 registration:

```
Enter command below:  
register  
[GIPSYNode] Sending NodeRegistrationDemand with destination: manager and signature: b88314da-10f8-4cac-a159-25b6f2964635  
[GIPSYNode] NodeRegistrationDemand sent, waiting for the result ...  
[GIPSYNode] Node registration completed!  
[GIPSYNode] Node started!  
[GIPSYNode] Waiting for a new system demand ...
```

(b)

Figure 23: Screenshots of the GMT console and the two GIPSY Node consoles

The screenshots shown in Figure 2.23(a) show two console windows, the upper console is the GMT console, the lower console is the console window of **Node 0** where the GMT was started. The consoles show that the GMT and the DST were allocated in **Node 0** following the command `start GMT GMT.config` as expected, and that **Node 0** was automatically registered as expected as well. Figure 2.23(b) shows the console output of **Node 1** who was registered into the GMT following the

command `register`, and the GMT console in Figure 2.23(a) also shows that **Node 1** was registered. These results indicate that the GIPSY instance bootstrap process and the node registration process were successful.

2.4.2 GIPSY Tier Allocation and Deallocation

Objective

To test if the GIPSY tier allocation and deallocation is working.

Scenario

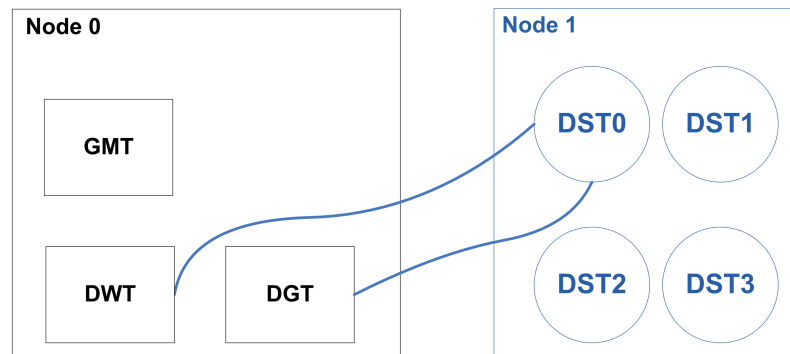


Figure 24: Test scenario for GIPSY tier allocation and deallocation

As shown in Figure 24, two computers are used as the GIPSY nodes in this test, i.e. **Node 0** and **Node 1** respectively. The precondition and the steps of this test are:

Precondition: **Node 0** has the GMT allocated and itself registered in the GIPSY instance bootstrap process, and **Node 1** is registered into the GMT as well.

Step 1: allocate 4 different DSTs in **Node 1** using 4 different configuration files in as shown in Figure 24 by entering the following commands into the GMT console (the meaning of these demands were explained in Section 2.3.3):

```
allocate 1 DST DSTProfiles/p9.config
```

```
allocate 1 DST DSTProfiles/p10.config
allocate 1 DST DSTProfiles/p11.config
allocate 1 DST DSTProfiles/p12.config
```

The expected results of this step are: the GMT console outputs that the 4 DSTs are allocated, and the console windows of the 4 DSTs appear in **Node 1**.

Step 2: allocate 1 DWT and 1 DGT in **Node 0** connecting to the first DST allocated in Step 1, as shown in Figure 24, by entering the following commands into the GMT console (the meaning of these demands were explained in Section 2.3.3):

```
allocate 0 DWT DWT.config 1
allocate 0 DGT sDGT.config 1
```

The expected results of this step are: the console window of **Node 0** outputs that the DWT and the DGT are allocated.

Step 3: deallocate the DWT and the DGT allocated in **Node 0** in Step 2, by entering the following commands into the GMT console (the meaning of these demands were explained in Section 2.3.3):

```
deallocate 0 DWT 0
deallocate 0 DGT 0
```

The expected result of this step is: the console window of **Node 0** outputs that the DWT and the DGT are deallocated.

Step 4: deallocate the 4 DST allocated in **Node 1** in Step 1, by entering the following command into the GMT console (the meaning of this demand was explained in Section 2.3.3):

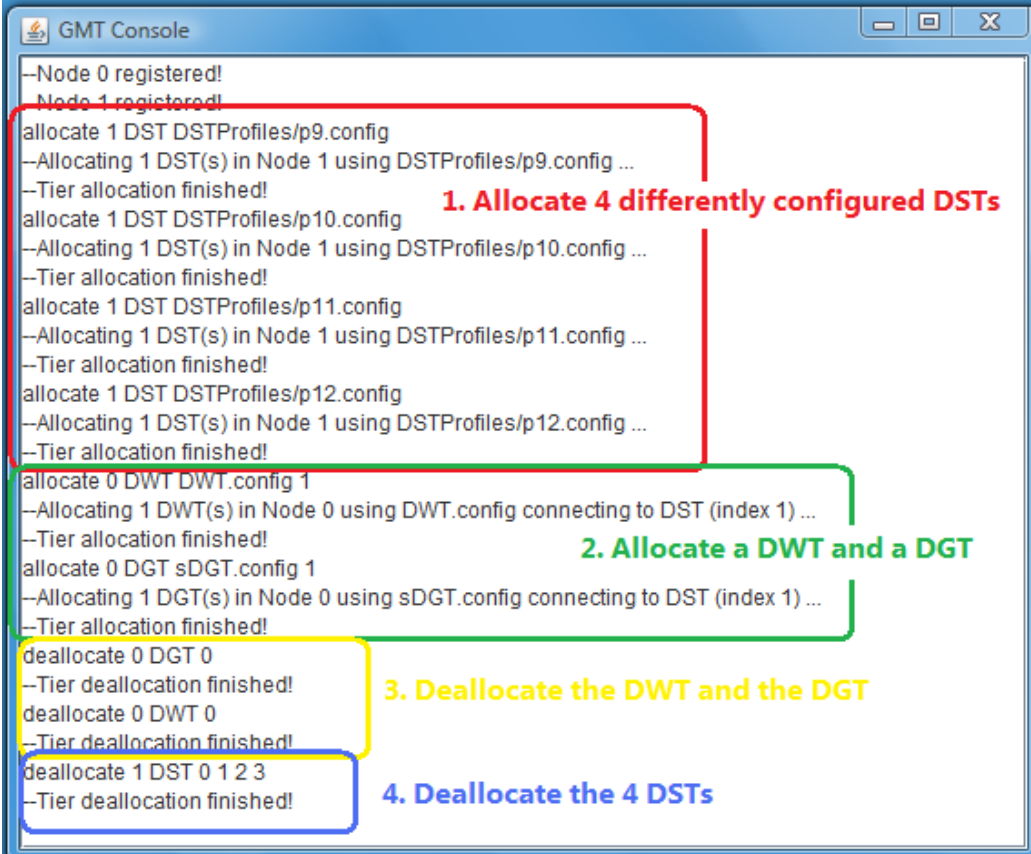
```
deallocate 1 DST 0 1 2 3
```

The expected result of this step is: the console window of **Node 1** outputs that the 4 DSTs are deallocated, and the console windows of the 4 DSTs are automatically closed.

Environment

The test was performed in two computers with the configuration specified in Table 1.

Result



```
GMT Console
--Node 0 registered!
Node 1 registered!
allocate 1 DST DSTProfiles/p9.config
--Allocating 1 DST(s) in Node 1 using DSTProfiles/p9.config ...
--Tier allocation finished!
allocate 1 DST DSTProfiles/p10.config
--Allocating 1 DST(s) in Node 1 using DSTProfiles/p10.config ...
--Tier allocation finished!
allocate 1 DST DSTProfiles/p11.config
--Allocating 1 DST(s) in Node 1 using DSTProfiles/p11.config ...
--Tier allocation finished!
allocate 1 DST DSTProfiles/p12.config
--Allocating 1 DST(s) in Node 1 using DSTProfiles/p12.config ...
--Tier allocation finished!
allocate 0 DWT DWT.config 1
--Allocating 1 DWT(s) in Node 0 using DWT.config connecting to DST (index 1) ...
--Tier allocation finished!
allocate 0 DGT sDGT.config 1
--Allocating 1 DGT(s) in Node 0 using sDGT.config connecting to DST (index 1) ...
--Tier allocation finished!
deallocate 0 DGT 0
--Tier deallocation finished!
deallocate 0 DWT 0
--Tier deallocation finished!
deallocate 1 DST 0 1 2 3
--Tier deallocation finished!
```

1. Allocate 4 differently configured DSTs

2. Allocate a DWT and a DGT

3. Deallocate the DWT and the DGT

4. Deallocate the 4 DSTs

Figure 25: GMT console output for tier allocation and deallocation

Figure 25 shows the screenshot of the GMT console, which clearly shows the GMT commands (lines that begin without hyphens) entered by the user and the corresponding outputs (lines that begin with two consecutive hyphens) produced by the GMT. The commands and their outputs corresponding to the 4 test steps are highlighted in the figure as well.

As expected for Step 1, the GMT console in Figure 25 and the corresponding console output of **Node 1** shown in Figure 28 indicate that the 4 DSTs were allocated in **Node 1**. The console windows of the 4 different DSTs are shown in Figure 26.

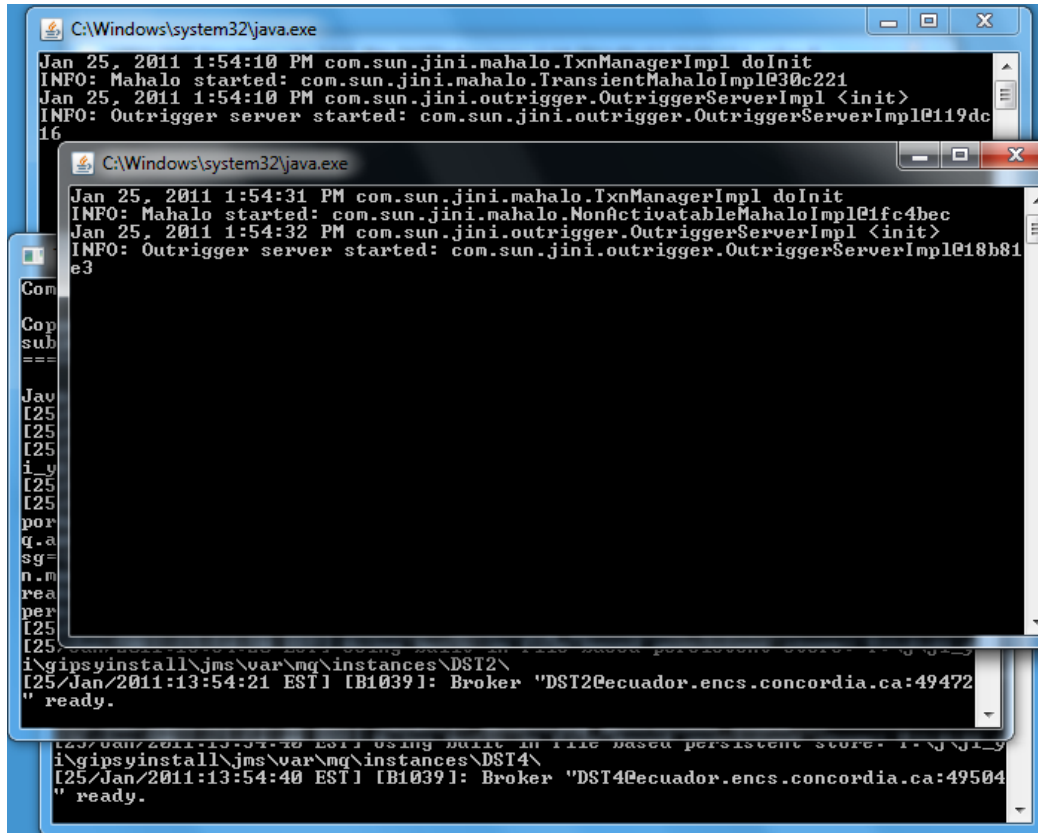


Figure 26: The four DSTs allocated

As expected for Step 2, the GMT console in Figure 25 and the corresponding console output of **Node 0** shown in Figure 27 indicate that 1 DWT and 1 DGT were allocated in **Node 0**.

As expected for Step 3, the GMT console in Figure 25 and the corresponding console output of **Node 0** shown in Figure 27 indicate that the DWT and the DGT allocated in **Node 0** were deallocated.

As expected for Step 4, the GMT console in Figure 25 and the corresponding console output of **Node 1** shown in Figure 28 indicate that the 4 DSTs allocated in **Node 1** were deallocated. Figure 28.

The results of this test shows that the GIPSY runtime system developed can successfully perform tier allocation and deallocation tasks as designed.

2.5 Summary

This chapter presents the design, implementation and tests of a deployable GIPSY runtime system using the multi-tier architecture, and shows that the GIPSY instance allows the user to register GIPSY nodes, allocate and deallocate GIPSY tiers among multiple networked computers, with which the scalability of the GIPSY runtime system can be assessed.

```
Enter command below:
start GMT GMT.config
[JiniDSTManager] Allocating new Jini DST ...
[JiniDSTManager] Jini DST started, PID = 4392
[GIPSYNode] Node started!
[GIPSYNode] Waiting for a new system demand
[GIPSYNode] Received a DWTAllocationRequest
[GIPSYNode] DWT0 was allocated and started!
[GIPSYNode] Sending TierAllocationResult enclosing tier registrations ...
[GIPSYNode] Waiting for a new system demand ...
[GIPSYNode] Received a DGTAllocationRequest
[GIPSYNode] DGT0 was allocated and started!
[GIPSYNode] Sending TierAllocationResult enclosing tier registrations ...
[GIPSYNode] Waiting for a new system demand ...
Demand payload prepared: 32768
[GIPSYNode] DGT0 deallocated!
[GIPSYNode] Sending deallocation result ...
[GIPSYNode] TierDeallocationResult sent!
[GIPSYNode] Waiting for a new system demand ...
[GIPSYNode] DWT0 deallocated!
[GIPSYNode] Sending deallocation result ...
[GIPSYNode] TierDeallocationResult sent!
[GIPSYNode] Waiting for a new system demand ...
```

Figure 27: The console output of Node 0

```

Enter command below:
register
[GIPSYNode] Sending NodeRegistrationDemand with destination: manager and signature: 48a12f42-c179-429b-a552-8649ea98e76d
[GIPSYNode] NodeRegistrationDemand sent, waiting for the result ...
[GIPSYNode] Node registration completed!
[GIPSYNode] Node started!
[GIPSYNode] Waiting for a new system demand
[GIPSYNode] Received a DSTAllocationRequest
[JiniDSTManager] Allocating new Jini DST ...
[JiniDSTManager] Jini DST started, PID = 1336
[GIPSYNode] DST0 was allocated and started!
[GIPSYNode] Sending TierAllocationResult enclosing tier registrations ...
[GIPSYNode] Waiting for a new system demand ...
[GIPSYNode] Received a DSTAllocationRequest
[JMSDSTWrapper] JMS Broker DST1 started at port 51009
[GIPSYNode] DST1 was allocated and started!
[GIPSYNode] Sending TierAllocationResult enclosing tier registrations ...
[GIPSYNode] Waiting for a new system demand ...
[GIPSYNode] Received a DSTAllocationRequest
[JiniDSTManager] Allocating new Jini DST ...
[JiniDSTManager] Jini DST started, PID = 4176
[GIPSYNode] DST2 was allocated and started!
[GIPSYNode] Sending TierAllocationResult enclosing tier registrations ...
[GIPSYNode] Waiting for a new system demand ...
[GIPSYNode] Received a DSTAllocationRequest
[JMSDSTWrapper] JMS Broker DST3 started at port 51062
[GIPSYNode] DST3 was allocated and started!
[GIPSYNode] Sending TierAllocationResult enclosing tier registrations ...
[GIPSYNode] Waiting for a new system demand ...
[GIPSYNode] DST0 deallocated!
[GIPSYNode] DST1 deallocated!
[GIPSYNode] DST2 deallocated!
[GIPSYNode] DST3 deallocated!
[GIPSYNode] Sending deallocation result ...
[GIPSYNode] TierDeallocationResult sent!
[GIPSYNode] waiting for a new system demand ...

```

Figure 28: The console output of Node 1

Chapter 3

Scalability Evaluation

3.1 Overview

Since GIPSY is a demand-driven system, the scalability of GIPSY runtime system is measured via software metrics regarding demand creation, propagation, storage and computation. As discussed in Section 1.3, the scalability of the GIPSY runtime system consists of the following four types of scalability. The metrics of these four types of scalabilities are described in Section 1.3.3.

- *Space scalability*: the ability of the GIPSY runtime system to store increasing amount of demands with tolerable memory usage. The judgment of “tolerable” may vary from situation to situation and from researcher to researcher. However in this thesis it is assumed that as long as a particular DST can store demands whose overall size at least equals the DST’s memory, the memory usage of the DST is “tolerable” since for the sake of demand storage it uses its memory well. The metric for this type of scalability is therefore the ratio between *total demand size* and *total DST memory size*. Ideally for a particular DST, if the ratio is equal to or greater than 1, the DST is considered space scalable since its memory requirement does not exceed the total size of its demand storage. In reality for the DSTs whose demand storage capacity is constrained by its memory, the ratio is expected to be less than 1 due to reasons such as

not all the objects stored in the memory of DSTs are demands. Also, given a *total DST memory size*, the *maximum demand size* of a single demand that can be stored into the DST is also of our interest.

- *Space-time scalability*: the ability of the GIPSY runtime system to maintain its performance measured in the *demand response time*, i.e. the *demand sending time* plus the *result reading time* experienced by a DGT when the amount of demands stored in the DST to which the DGT connects is increasing. For a particular DST, if the *demand response time* increases with the amount of demands stored in the DST, the DST is considered not space-time scalable because its performance is undermined by the increase of the *number of demands*; otherwise it considered as space-time scalable.
- *Structural scalability*: the ability of the GIPSY runtime system to allocate more GIPSY tiers over more GIPSY nodes (Section 1.2.2). This type of scalability is assessed by estimating the *maximum node/tier registrations* that a typical GMT can store and the *maximum DST connections* supported by the DSTs. DSTs that can support more connections are considered more structurally scalable.
- *Load scalability*: the ability of the GIPSY runtime system to achieve a maximum demand-processing *throughput* that is able to increase proportionally with the number of tiers that perform the demand computation. Since multiple tiers can be allocated within one GIPSY node, taking a particular type of procedural demands as an example, the minimum number of DWTs to be deployed within one GIPSY node to sufficiently utilize its computation power shall be determined experimentally via the metric *throughput* versus *number of tiers* that perform the computation, and with this minimum number of DWTs deployed in each GIPSY node, how the *throughput* changes with the total *number of DWTs* allocated in all the GIPSY nodes to perform the demand computation is used to assess the load scalability.

To test the above four types of scalability, a DGT simulator is used as the test driver to generate and send demands of different types and features, because the simulator provides more flexibility as it can easily generate different types of demands with different features to test different aspects of the GIPSY runtime system. For example, with the simulator it is easy to generate any number of demands of any size and computation complexity, whereas with a real DGT a special GIPSY program has to be designed to achieve a similar effect, which is inflexible and inefficient for testing purpose. Another reason is that the DGT can only test the entire demand migration process including the DGT sending demands, the DWT receiving and computing demands, and finally the DGT reading the computed results; in contrast with a simulator, the demand sending time and result reading time can be isolated via a tester thread that only sends demands or only reads demands for a better separation of concern.

Based on the four types of scalability and their metrics described above and with the use of the DGT simulator, the following sections present the assessment of the four types of scalability of the GIPSY runtime system respectively.

3.2 Space Scalability

In a GIPSY runtime system, demands are transferred via DSTs and are stored in DSTs for potential reuse. Therefore when a DST with limited memory is facing increasing amount of demands to store, how many demands it can store and if it uses its memory well for demand storage is covered by the space scalability of the GIPSY runtime system.

As introduced in Chapter 2, both the Jini and the JMS versions of the DST allow the user to use configuration settings to enable or disable persistence. Once crashed and restarted, a non-persistent or transient DST will lose its state because it stores demands in memory, whereas a persistent DST can recover its previous state from its persistent storage such as log files.

For a transient DST, either the Jini or the JMS version, since it runs as a JVM

(Java Virtual Machine) process and stores demands in its memory, its demand storage capacity is bound to be limited due to the following reasons:

1. Demands are Java objects stored in the heap of the JVM process, therefore the demand storage is constrained by the size and division of the heap. The JVM implementation shipped with the Java SE platform, the HotSpot VM (Virtual Machine) divides the heap into 3 generations: the young generation, the tenured (or old) generation and the permanent generation [49] as shown in Figure 29. The young generation is further divided into one *eden space* and two *survivor spaces*. Most objects are allocated in the *eden space* initially and are eventually copied to the old generation if they survive some rounds of garbage collections in the young generation. If the objects are too large to be allocated in the young generation, they may be allocated in the old generation directly. The combination of the young and the old generations is referred as the *heap memory* [50]. The permanent generation stores JVM metadata such as the classes and methods loaded, and although logically part of the heap, it is regarded as part of the *non-heap memory* [50]. The sizing of the young and the old generations affects the maximum amount of demands a DST can store. For example, given a demand size of 100 MB and a total *heap memory* size of 256 MB, if the *eden space* of the young generation has 125 MB free memory and the old generation also has 125 MB free memory, the overall *heap memory* can store up to two 100 MB demands with one demand per generation; whereas if the free space in the *eden space* is 80 MB and that in the old generation is 170 MB, only one 100 MB demand can be stored in the old generation, because after storing this demand, neither the young and the old generation can store another 100 MB demand, although the sum of their free memory space exceeds 100 MB.
2. Regardless the heap generations, the maximum overall *heap memory* size cannot exceed the maximum virtual memory address space per process. Such maximum per-process address space is platform dependent: in Windows the default

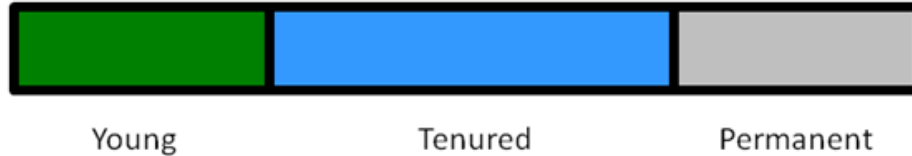


Figure 29: JVM heap generations

maximum memory that can be used by a process is 2 GB [51]; whereas in 32-bit Solaris a process can have 4 GB maximum memory space [52]. However, the maximum virtual memory space per process cannot be completely reserved by Java heap due to several reasons such as available swap space [53], memory reserved for code cache, stacks, as well as VM overhead such as necessary system and VM libraries [52]. The maximum heap size in 32-bit Windows is approximately 1.4 GB to 1.6 GB [52].

3. Not all the objects stored in the JVM heap of a DST are demands: some are objects automatically created by the DST to maintain its operation, for example the thread objects created to handle connections. The more memory occupied by these objects, the less heap space left for demands.

As to a persistent DST, if its persistent storage is for backing up its memory storage only, i.e. for the logging purpose only, then its storage capacity is limited by its memory; whereas if its persistent storage can store extra demands beyond its memory limit, then its storage capacity is limited by other constraints in such as the relevant configuration settings and the size of available hard disk space. The tests described in Section 3.2.1 figure out which DST can use its persistent storage to store demands beyond its memory limit, and which DST uses its persistent storage for the logging purpose only.

To measure the total size of all the demands stored in a DST, a special demand that carries an array of raw bytes as payload was designed. Given a DST that stores demands carrying the same payload, the overall payloads of all the demands stored in the DST is used as the *total demand size*, and the heap memory size of the DST is used as the *total DST memory size*. Based on this approach, the following tests

were designed to assess the space scalability of the GIPSY runtime system. For each one of the DSTs in question, i.e. the transient Jini DST, the transient JMS DST, the persistent Jini DST and the persistent JMS DST, the first test investigates the relation between the maximum *total demand size* that the DST can store and its *total DST memory size*, and the second test investigates the maximum size of a single demand that the DST with a particular heap memory size can store.

3.2.1 Space Scalability Test: Demand Storage

Objective

To test if the DSTs of different types and persistence settings are space scalable by finding the relation between the *total demand size* and the *total DST memory size* for each DST.

Scenario

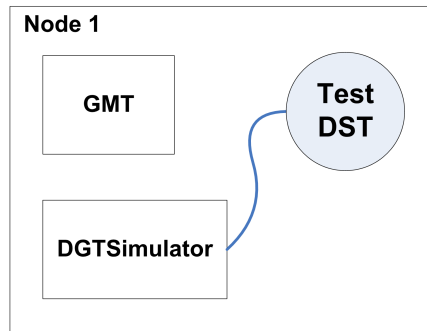


Figure 30: Demand storage test deployment

12 DST profiles are set up to test the DSTs of different types, different persistence settings and with different heap memory sizes as listed in Table 3. For each DST profile, a single GIPSY node is used to perform this test as illustrated in Figure 30: within the GIPSY node managed by the GMT, a DST is allocated using the DST profile and a DGT simulator (see Section 3.1) is allocated to feed this DST with demands carrying the same particular payload until the DST throws out-of-memory error. The maximum number of successfully stored demands is recorded and

is multiplied by the demand payload to compute the *total demand size*. The relation between the maximum *total demand size* and the *total DST memory size*, i.e. the heap memory size, is used to assess the DST’s space scalability. All the experiments are done in triplicate.

Environment and Parameters

The test was performed in a computer with the hardware and operating system environment specified in Table 2.

OS Name	Microsoft Windows 7 Enterprise
Version	Version 6.1.7600 Build 7600
System Type	X86-based PC
Processor	Intel(R) Core(TM)2 CPU 6300 @ 1.86GHz, 1862 Mhz, 2 Core(s), 2 Logical Processor(s)
Installed RAM	2.00 GB
Total RAM	2.00 GB
Available RAM	1.06 GB
Total Virtual Memory	4.00 GB
Available Virtual Memory	2.58 GB
Page File Space	2.00 GB

Table 2: Hardware and operating system environment

The 12 DST profiles are listed in Table 3. All the DST profiles share the same fixed heap space partition ratio so that their memory usage is predictable. The ratio between the young and old generation is 1:1 and the ratio between the *eden space* and a single *survivor space* is 1022:1 to allow most of the entire heap memory to be used by the DST, since JVM deliberately leaves one of the two *survivor spaces* empty for garbage collection purpose [49]. The first heap size tested is 64MB, which is the default maximum heap size of non-server class machines [49], then the heap size is double to 128M, and finally doubled to 256MB. For each heap size, the DSTs of different types and persistence settings are tested to find the maximum amount of demands they each can store. Since persistent JMS DSTs can store messages in hard disk [48], their overall messages sizes are set to be 50

The demand payload selected is 32 KB, which is the size of the minimum heap

Profile #	Type	Persistence	Heap Size and Partition	Other
1	Jini	Transient	64 MB heap memory with young : old generations = 1:1 eden : survivor spaces = 1022:1	
2	JMS	Transient	Same as Profile 1	
3	Jini	Persistent	Same as Profile 1	
4	JMS	Persistent	Same as Profile 1	Overall message size = 96 MB
5	Jini	Transient	128 MB heap memory with young : old generations = 1:1 eden : survivor spaces = 1022:1	
6	JMS	Transient	Same as Profile 5	
7	Jini	Persistent	Same as Profile 5	
8	JMS	Persistent	Same as Profile 5	Overall message size = 192 MB
9	Jini	Transient	256 MB heap memory with young : old generations = 1:1 eden : survivor spaces = 1022:1	
10	JMS	Transient	Same as Profile 9	
11	Jini	Persistent	Same as Profile 9	
12	JMS	Persistent	Same as Profile 9	Overall message size = 384 MB

Table 3: The 12 DST profiles

space fragmentation that was set so far to simplify memory relevant calculation and comparison.

Results

For each DST profile, the maximum number of demands and the calculated *total demand size* is shown in Table 4. A graphical representation illustrating the relation between the maximum *total demand size* and the *total DST memory size*, i.e. the heap memory size, is in Figure 31.

Figure 31 shows that the *total demand size* of persistent JMS DST was not constrained by its heap size, rather, it was constrained by the *overall message size* specified in its profiles for the simulation of limited hard disk space. This verifies that

Demand Payload	DST Profile #	Max. Amount of Demands	<i>total demand size</i> (MB)
32 KB	1	1899	59.36
32 KB	2	1785	55.80
32 KB	3	1006	31.44
32 KB	4	2917	91.16
32 KB	5	3822	119.44
32 KB	6	3628	113.39
32 KB	7	2032	63.51
32 KB	8	5834	182.31
32 KB	9	7674	239.83
32 KB	10	7284	227.64
32 KB	11	4108	128.39
32 KB	12	11669	364.66

Table 4: Demand storage test result

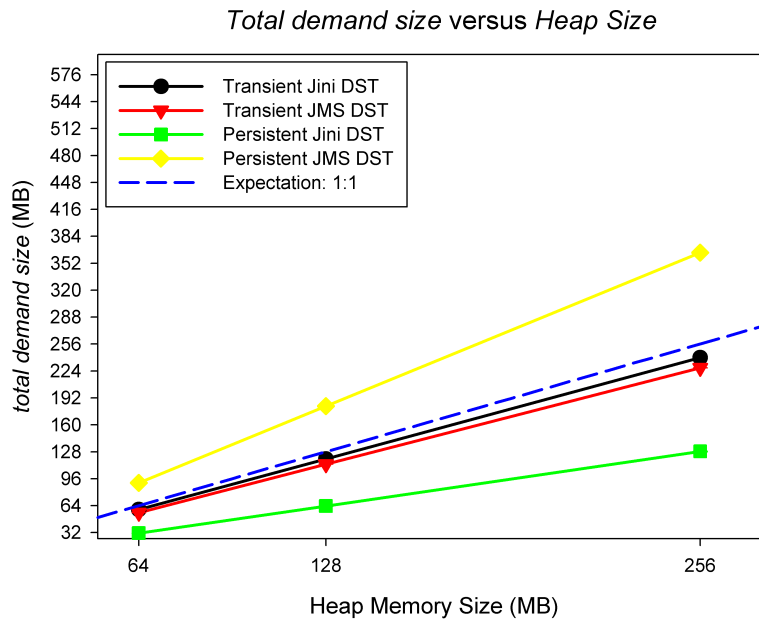


Figure 31: Maximum total demand size versus heap size

the Sun Message Queue can indeed swap messages from memory to its persistent storage when its memory usage reaches certain level [48]. Due to this feature, the persistent JMS DST is the most space scalable because its demands storage is not constrained by its memory and in this test, the ratio between *total demand size* and *total DST memory size* was 1.42, the highest among all the DSTs. In contrast, the maximum *total demand size* of the transient Jini DST, the transient JMS DST and the persistent Jini DST was confined by their heap memory, and their ratio between their *total demand size* and *total DST memory size* were 0.93, 0.88 and 0.49 respectively, which means that the transient Jini and the transient JMS DST can use most of their memory for demand storage, whereas the persistent Jini can only use half of its memory for demand storage. The 0.49 memory usage for demand storage of the persistent Jini DST implies that firstly the persistent Jini cannot store additional demands beyond its memory limit into its persistent storage, i.e. its persistent storage is for logging purpose only, and secondly the significant reduce in its maximum *total demand size* was caused by the persistent service, which was the only difference between the persistent Jini DST and the transient Jini DST. Since most memory of the transient Jini and the transient JMS DSTs can be used for demand storage, they are considered space scalable, whereas the persistent Jini DST is considered space unscalable due to that fact that only 0.49 of its memory can be used for demand storage, which is not resource efficient.

3.2.2 Space Scalability Test: Maximum Demand Size

Objective

For each one of the transient Jini DST, transient JMS DST, persistent Jini DST and persistent JMS DST, find the maximum demand size of a single demand that the DST can store, and find how many such demands that the DST can store.

Scenario

A single GIPSY node is used to perform this test, as illustrated in Figure 30. The DST under test is allocated using one of the profiles specified in the next paragraph, and a DGT simulator (see Section 3.1) is allocated to feed the DST with demands of various sizes to find the maximum demand size that the DST can store. In this way the maximum demand sizes supported by the DSTs of different types and persistence settings can be obtained.

Environment and Parameters

The hardware and operating system environment is the same as that in the previous test as shown in Table 2. The DSTs are started using the profile number 9, 10, 11, 12 that were used in the previous test, all of which use the 256 MB heap memory size with 1:1 young-old generation ratio to allow the DGT simulator to send necessarily large demands without running out of memory. In this test, 1MB is chosen as the magnitude of demand size to ease the maximum-size finding process.

Results

The maximum demand size that could be stored in each DST as well as the maximum amount of such demands that could be stored in the DST is listed in Table 5, and the bar chart showing the relation between the *maximum demand sizes* and the heap memory size is in Figure 32.

Profile #	Max. Demand Payload	Max. Amount of Demands
9	127 MB	1
10	127 MB	1
11	63 MB	1
12	126 MB	> 100

Table 5: Maximum demand size test result

The results of this test show that maximum demand payload supported in the transient Jini DST, the transient JMS DST and the persistent JMS DST were around 127 MB, which is reasonable because the largest heap partition, i.e. the old generation

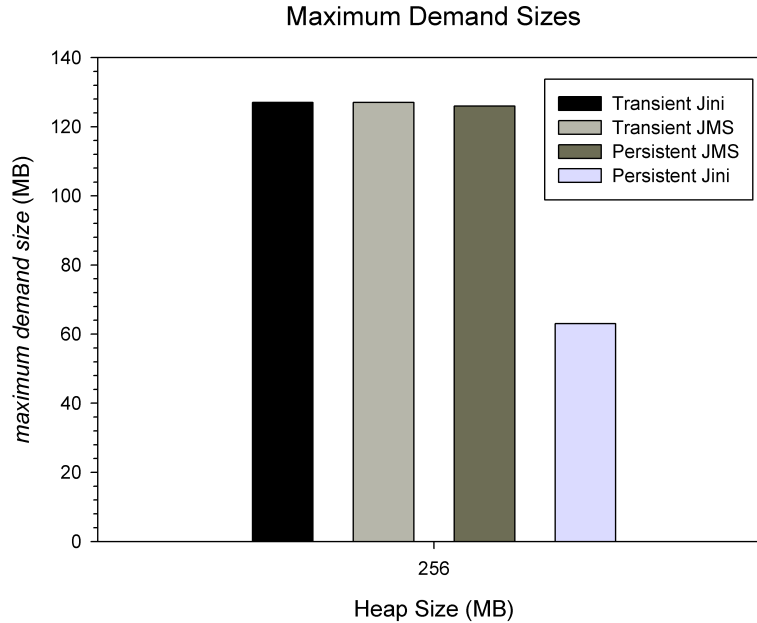


Figure 32: Maximum demand size for 256 MB heap

in their heaps was 128 MB. Also, the persistent JMS DST could store much more demands of the maximum size since it can swap demands into its persistent storage beyond its memory limit as it did in the previous test, which is judged more space scalable. It is also shown that the persistent Jini DST did not support demands as large as those supported by other DSTs, similar to what it did for 32-KB payload demands.

The reason why this test is necessary is that in reality a demand could be very large. For example, since all the runtime resources are contained in the GEER (see Section 1.1.2), the GEER transferred by a resource demand could be very large depending on how many runtime resources are contained. Also the data sets required by the procedural demands could be very large; for example, for some image or audio processing demands, some large image or audio files could be used as the parameters or return values of the demands. Therefore this test of the maximum size of a single demand that a DST can store is of real importance.

3.2.3 Summary

In this section the space scalability of GIPSY runtime system was assessed by testing how the different types of DSTs with different persistence settings and limited memory handled increasing demand storage. Also the maximum demand sizes that different DSTs supported were compared. It was discovered that the transient Jini DST, the transient JMS DST and the persistent JMS DST are space scalable as they can store increasing amount of memory while making good use of their memory, as discussed in the test results in this section; in contrast the persistent Jini DST can only use approximately half of its memory to store demands when storing increasing amount of demands, which is considered not space scalable.

3.3 Space-Time Scalability

The space-time scalability of GIPSY runtime system mainly concerns how the amount of demands stored in DSTs affects *demand response time* of demand migration consisting of the *demand sending time* and the *result reading time* experienced by the DGT. The *demand sending time* is the time spent in sending a demand into the DST, and the *result reading time* is the time spent in reading a demand whose state *matches* the *computed* state and whose signature *matches* the demand signature in question from the DST. Since the amount of demands stored in a DST may affect both its memory usage and its speed in demand signature matching (when returning a result to the DGT), the following tests were designed: the first test investigates how the increase in the total size of demand storage, i.e. the memory usage, affects the overall *demand response time*, and the second test isolates the *demand sending time* and the *result reading time* out to investigate how the increasing amount of demands affects these two times.

3.3.1 Space-Time Scalability Test: Memory Impact

Objective

For each of the space-scalable DSTs, test how the *demand response time* is affected by the increase in the total size of all the demand stored in the DST.

Scenario

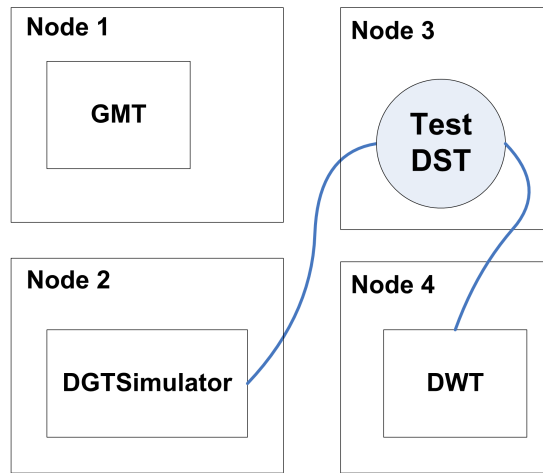


Figure 33: Space-time scalability test deployment: memory impact

In this test the space scalable DSTs, i.e. transient Jini DST, the transient JMS DST and the persistent JMS DST are tested. A GIPSY instance consisting of four GIPSY nodes is used to perform this test, as illustrated in Figure 33, so that each tier is running within its own GIPSY node to separate their CPU usage. In this test, a GMT is allocated in Node 1 to control tier allocations; a test DST is allocated in Node 3; a DGT simulator (see Section 3.1) is allocated in Node 2 to continuously send a unique *pending* demand to the DST and read the *computed* result back before sending the next demand; and a DWT is allocated in Node 4 to continuously receive *pending* demands from the DST and return the *computed* result to the DST immediately. The time taken by each demand sending and result reading operation is recorded until the DST can no longer store any more demands. Each demand is carrying a 32 KB payload to measure the DST's memory usage. All the experiments are done in

triplicate.

Environment and Parameters

The four GIPSY nodes used in this test all have the same hardware and operating system environment as shown in Table 2. The DSTs are started using the profile number 9, 10, and 12 that were used in the space-scalability tests described in Section 3.2.1 to start the transient Jini DST, transient JMS DST and persistent JMS DST respectively with all the DSTs having a 256 MB heap size.

Results

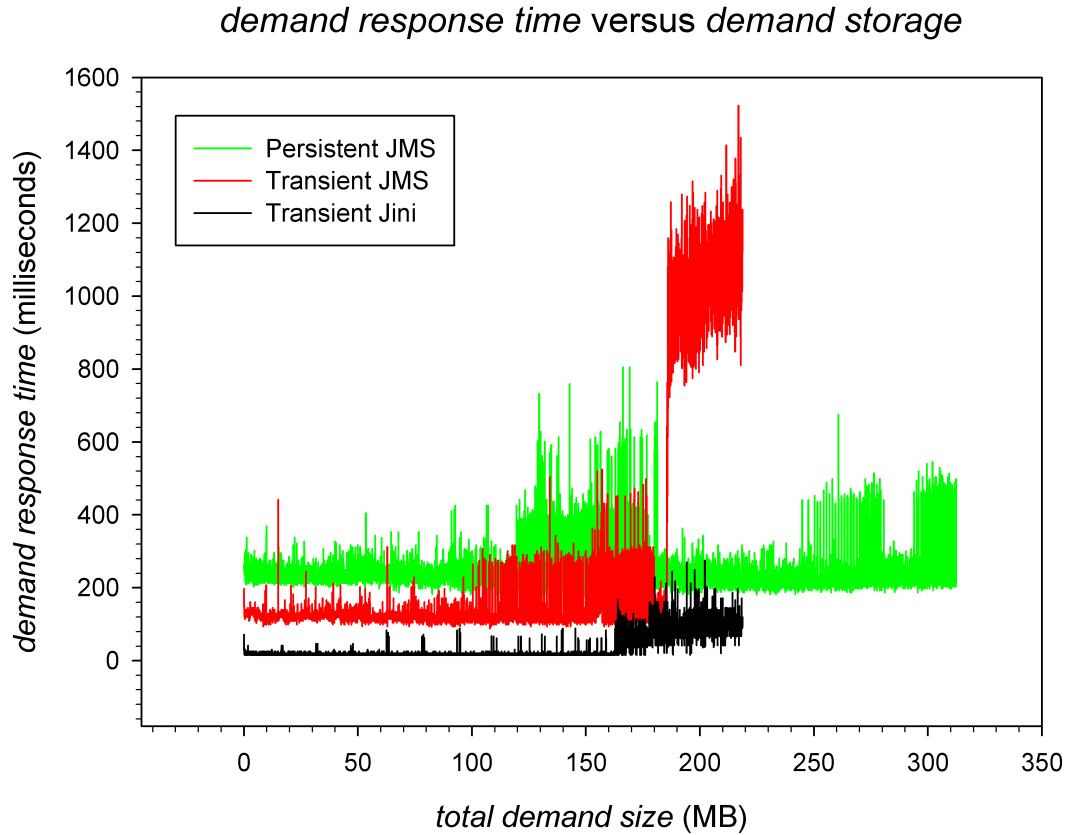


Figure 34: Response time versus demand storage

Figure 34 shows the *demand response time* versus the *total demand size* stored in the DST. It is shown that in the beginning, the *demand response time* provided by all the DSTs were relatively stable, and the transient Jini DST was faster than

the transient JMS DST that was even faster than the persistent JMS DST, which is can be explained by the fact that the JMS DSTs require more complicated services than the Jini DST, such as memory monitor and control, and that the persistent JMS needs extra time for the persistent storage service. However, as the demand storage increased, their response times changed due to the way how they use and manage memory:

1. The Jini DST relies on garbage collection to manage its memory. When its young generation was full and the old generation was initially empty, objects were eventually copied to the old generation via minor garbage collections; therefore in the beginning, there were relatively short pause time occasionally. As the free space in the old generation decreased, it could no longer accommodate all the living objects to be copied from the young generation, therefore the entire heap was collected and this major garbage collection caused longer pause time from the time when the demand storage was approximately 60 MB. As the demand storage approached 160 MB, the old generation became full and the young generation was also being filled up, therefore the frequency of major garbage collections increased until its maximum was reached, resulting in final part of the curve for the Jini DST.
2. The transient JMS DST provides additional memory management in addition to the heap garbage collection. For example, Sun Message Queue has four memory resource states, *green*, *yellow*, *orange* and *red* indicating the percentage of the heap memory usage of 0%, 80%, 90% and 98% respectively, and reacts to these memory resource states accordingly [48]. As the demand storage increased, the transient JMS DST went through similar garbage collection stages as the Jini DST. However, when it used approximately 80% of its memory (180MB divided by 227.64 MB, the maximum demand storage shown in Section 3.2.1), it was in the *yellow* state and began to search all the messages stored so far to swap all the persistent messages into persistent storage, although as a transient JMS it had no persistent messages to store, while the major garbage collections were

very frequent. This explains why after 180MB the *demand response time* were extremely long and increasing with the amount of demands stored.

3. The persistent JMS DST has the same memory management mechanism with transient JMS DST, so its *demand response time* shared a similar pattern to that of the transient JMS DST before its memory usage reached the *yellow* state. However, as all its messages were persistent messages, it was able to swap all the persistent messages into its persistent storage to free its memory, therefore from that point its *demand response time* fell back to the previous pattern.

Therefore when facing increasing demand storage, all the DSTs were slowed down when their memory went critical, and the persistent JMS DST could recover its memory by swapping persistent message into its persistent storage to reduce its memory usage.

3.3.2 Space-Time Scalability Test: Signature Matching Speed

Objective

For each space-scalable DST, test how the amount of demands affects the demand signature matching speed in the result reading process.

Scenario

In this test demands that each carries a string representation of 128-bit universally unique identifier are used to make the signature *matching* time more observable. These demands carry no payload so that their impact on memory is less significant. A GIPSY instance consisting of three GIPSY nodes is used to perform this test as illustrated in Figure 35. Similar to the previous space-time scalability test, each tier is running in its own GIPSY node to avoid interference in CPU usage. To separate the *demand sending time* and *result reading time* out of the overall demand migration

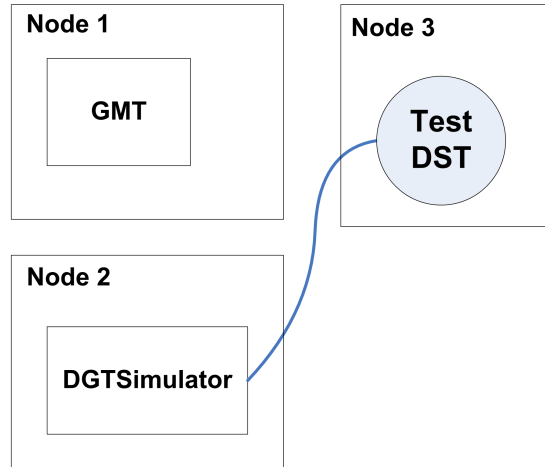


Figure 35: Space-time scalability test deployment: signature matching speed

process for the investigation of demand signature matching speed, the DGT simulator (see Section 3.1) uses a special tester thread to send and receive demands in 20 batches with each batch containing 500 such demands, so totally there are 10,000 demands tested, and each batch of demands is sent to and read from the DST synchronously before the sending and receiving the next batch. The *demand sending time* and *result reading time* of each demand is recorded. All the experiments are done in triplicate.

Environment and Parameters

The three GIPSY nodes used in this test all have the same hardware and operating system environment as shown in Table 2. The DSTs are started using the profile number 9, 10, and 12 that were used in the space-scalability tests described in Section 3.2.1 to start the transient Jini DST, transient JMS DST and persistent JMS DST respectively with all the DSTs having a 256 MB heap size.

Results

Figure 36 shows all the *demand sending time* recorded versus the amount of demands stored in each DSTs and Figure 37 shows the average *demand sending time* per batch (500 demands) versus the amount of demands stored in each DST. The curves of the

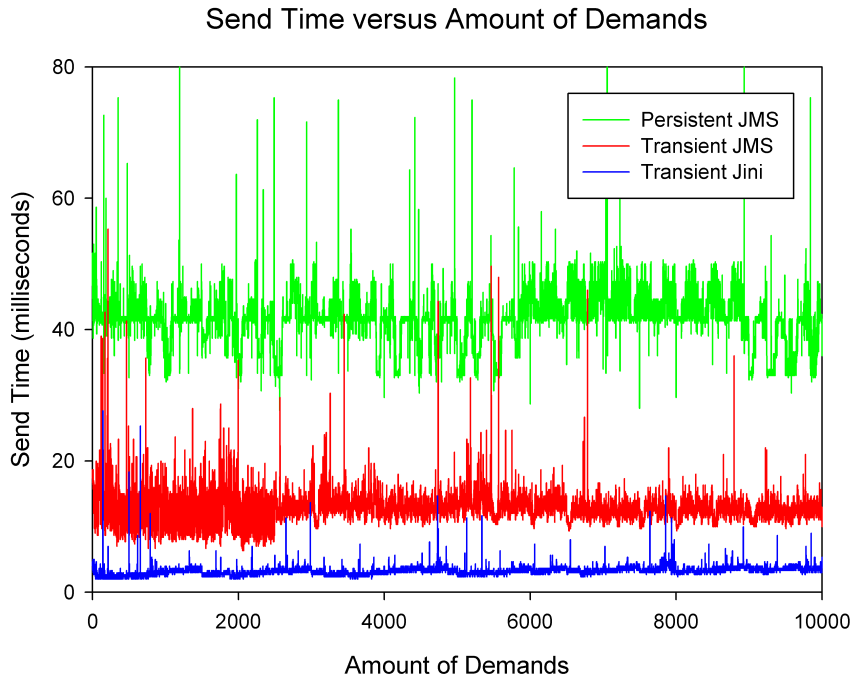


Figure 36: Demand sending time versus the amount of demands stored in the DST

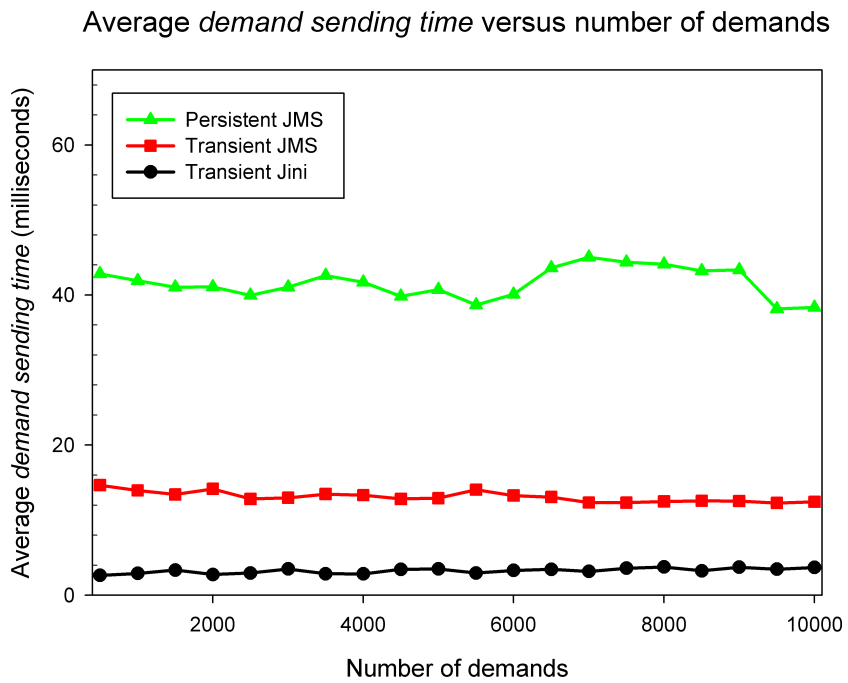


Figure 37: Average demand sending time versus the amount of demands stored in the DST

demand sending time and the average *demand sending time* shows that in this test scenario for each DST, the time used in sending demands to the DST was not increasing with the amount of demands stored in the DST, which is consistent with the trend of their *demand sending time* and *result reading time* shown in Figure 34 before their memory went critical. Also the transient Jini DST was the faster than the transient JMS DST that was approximately twice as fast as the persistent JMS DST.

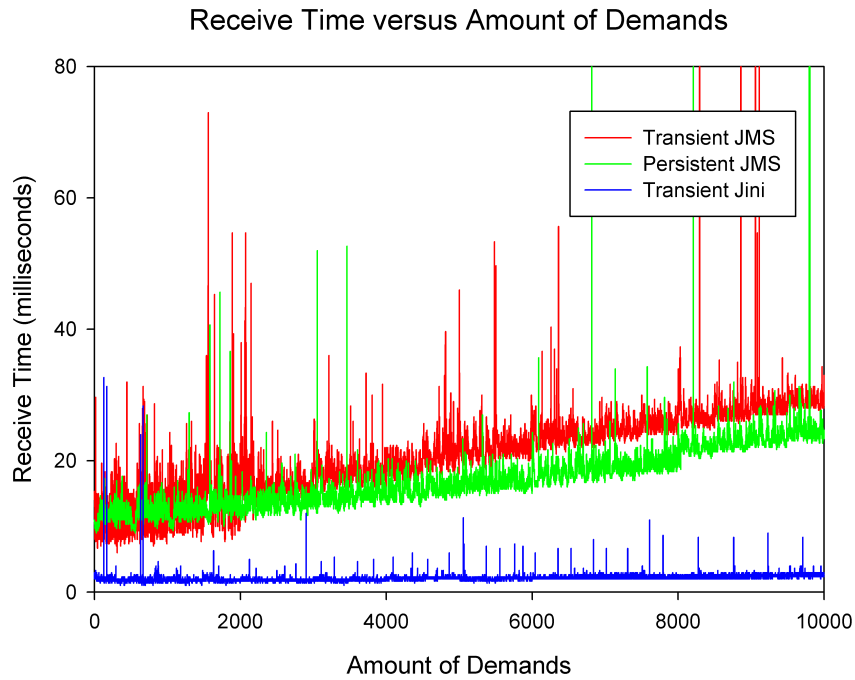


Figure 38: Result reading time versus the amount of demands stored in the DST

Figure 38 shows the recorded *result reading time* versus the amount of demands stored in each DSTs and Figure 39 shows the average *result reading time* versus the amount of demand stored in each DSTs. It is shown that in this test scenario, the average time spent in receiving demands from the transient Jini DST was not significantly increasing with the amount of demands stored in the DST, compared to the JMS DSTs whose demand reading time increased almost linearly with the amount of demands stored. Therefore in the sense of demand signature matching speed versus the amount of demands, the transient Jini DST is space-time scalable since its performance is not undermined by the amount of demands stored, whereas

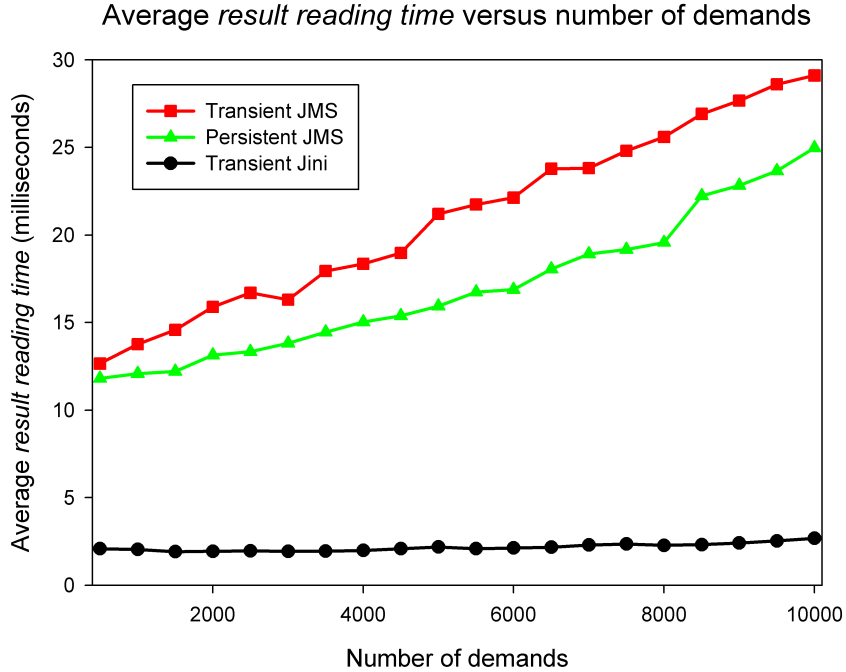


Figure 39: Average result reading time versus the amount of demands stored in the DST

the JMS DSTs are not space-time scalable since their performance are undermined by the amount of demands stored.

The average *demand sending time* and *result reading time* measured in this test can be used to estimate the communication cost associated with each DST under test, because the demand used to perform this test has no payload and has no computation requirements. For the transient Jini DST, the average time spent in sending such an *empty* demand is around 2.63 milliseconds, and the average time spent in reading the demand is around 2.09 milliseconds. For the transient JMS DST, the average time spent in sending an *empty* demand is around 14.67 milliseconds, and the average time spent in reading the demand is around 12.65 milliseconds. For the persistent JMS DST, the average time spent in sending an *empty* demand is around 42.82 milliseconds, and the average time spent in reading the demand is around 11.82 milliseconds. Such communication cost includes the time spent in setting up and tearing down the connections, the time in delivering the data across the network, and the time in other aspects such as ensuring reliable delivery. Further investigation of

the communication cost shall be done in future work.

3.3.3 Summary

In this section the space-time scalability of GIPSY runtime system was tested by investigating how the increase in the total size of the demand storage affects the *demand response time* and how the amount of demands stored in the DST affects the demand signature matching speed. It is discovered that as the total size of the demand storage increased, the *demand response time* provided by the Jini DST was the shortest, whereas that provided by the transient JMS DST was longer, and that of the persistent JMS DST was the longest; however, all the DSTs slowed down when their memory went critical, and the persistent JMS DST was able to recover its memory by moving messages to persistent storage. As to demand signature matching speed, the *result reading time* of the transient Jini DST was not increasing with the amount of demands stored, where as the result reading time provided by the JMS DSTs was linearly increasing with the amount of demands stored. Based on these results, the Jini DST is considered space-time scalable, whereas the JMS DSTs are not space-time scalable.

3.4 Structural Scalability

The structural scalability of GIPSY runtime system is the ability of the system to grow larger by allocating more tiers and registering more nodes, and the multitier architecture of GIPSY runtime system was designed to have this kind of scalability. For example, when the existing DSTs can no longer store more demands, GIPSY runtime system can easily deal with this situation by allowing the user to allocate more DSTs via GMT; similarly when all the DWTs are busy while there are still many procedural demands pending, more DWTs can be allocated to process these *pending* demands in parallel so that they demands may be returned to the DGTs that generated these demands sooner. In this way the GIPSY runtime system can gracefully deal with increasing work load by growing larger.

The assessment of this kind of scalability concerns how large a GIPSY runtime system can grow, i.e. how many tiers it can allocate and how many nodes can be registered. Since all the nodes/tiers are registered into and managed by the GMT, these questions can be converted to how many node/tier registrations a GMT can store and manage. Besides, since the DGTs and the DWTs are interconnected by the DSTs, we are also interested in how many DGTs or DWTs that a single DST can support. However, there are no definite answers to these questions because:

1. Since the current GMT implementation runs within a GIPSY node process and stores all the node/tier registrations in its memory, the amount of memory that can be used to store node/tier registrations varies with the overall heap memory size and with the amount of memory used by other tiers allocated within the same GIPSY node process.
2. Since each DST is a JVM process and launches threads to handle each connection, the maximum amount of threads is affected by the amount of heap memory available to the thread objects as well as the amount of non-heap virtual memory available to the threads stacks.

Due to the discussions above, the following tests were designed to investigate the structural scalability of the GIPSY runtime system.

3.4.1 Structural Scalability Test: GMT

Objective

For a typical GMT, test how many node/tier registrations it can support to estimate the maximum size of GIPSY instance it supports.

Scenario

To test how many tier registrations a typical GMT can store, two GIPSY nodes are used as illustrated in Figure 40. The word "typical" here means that the GMT is the only tier running Node 1, and the JVM settings of Node 1 are the default

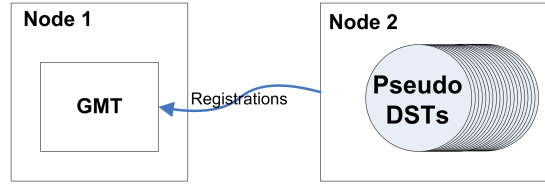


Figure 40: Structural scalability test deployment: GMT

client-class JVM settings for the 32-bit Windows system, as the worst case scenario. Node 2 is used to allocate pseudo DSTs that create DST registrations containing all the necessary information and configurations but without creating the real DST processes, so that Node 2 can easily allocate tens of thousands of pseudo DSTs and send the registrations to the GMT without creating too many processes to run out of resources. The reason why the DST registration is used rather than other node/tier registrations is that: typically a DST registration contains more information such as the exported TA configuration than other node/tier registrations, therefore is more representative.

Environment and Parameters

The hardware and operating system environment of each GIPSY node is shown in Table 2. The heap size of the GIPSY node process that runs the GMT is 64MB, the default JVM heap size in a client-class machine, and no other heap setting is required for this JVM process. All the experiments are done in triplicate.

Results

The average DST registrations that the GMT can store is 44,767, which means that if the GMT does nothing but stores node/tier registrations, the *maximum node/tier registrations* that it can support is approximately 44,000. In reality this number is expected to be smaller because extra memory is consumed for management routines such as tier allocations. By enlarging the heap size, the GMT could support more node/tier registrations; however, it is suggested to store all the node/tier registration in a more scalable database in the future to remove the memory constraint.

3.4.2 Structural Scalability Test: DST

Objective

For each space-scalable DST, test how many DWTs/DGTs that a DST can support.

Scenario

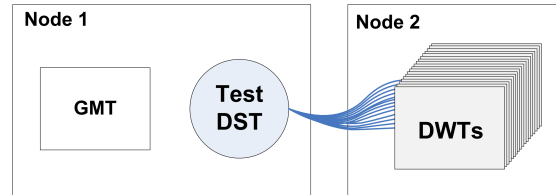


Figure 41: Structural scalability test deployment: DST

The transient Jini DST, the transient JMS DST and the persistent JMS DST are tested in this test, and DWTs are used to connect to the DSTs. Two GIPSY nodes are used to perform this test as illustrated in Figure 41, with the DST under test is allocated in Node 1, and the DWTs connecting to the test DST allocated in Node 2. For each test DST, as many as possible DWTs are allocated until the DST can no longer support more DWTs. The memory and the thread usage of the DST is observed and recorded via JConsole: before connecting the DWTs to the DST, the DST is garbage collected and its minimum memory and thread usages are recorded; when the DST can no longer support more DWTs, the DST is garbage collected and its minimum memory and thread usage are recorded again. We assume that the differences between the two minimum memory and thread usages are used by the DST to handle connections. Also, to investigate the impact of the stack size of DST threads, 3 stack sizes are tested. Based on the observation in this test, an estimated maximum number of connections that the DST can support is given.

Environment and Parameters

The hardware and operating system environment of each GIPSY node is shown in Table 2. The 3 stack sizes tested are 320 KB, 672 KB and 1 MB, among which 320

KB is the default JVM stack size in 32-bit Windows [53], 1 MB is the default native stack size in Windows [54], and 672 KB is in the middle. Also in this test both DSTs use the 256 MB heap setting that were used in previous tests to ensure that they have sufficient memory.

Results

The results of this test are presented in Table 6. And a bar chart based on the test results is shown in Figure 42.

DST Type & Persistence	Stack Size	Memory Before Test	Memory After Test	Min. Thread	Max. Thread	Max. DWT Allocation
Transient Jini	320 KB	3.01 MB	124.54 MB	44	4855	4809
Transient Jini	672 KB	3.65 MB	57.90 MB	51	2267	2219
Transient Jini	1 MB	3.04 MB	39.11 MB	51	1560	1510
Transient JMS	320 KB	3.46 MB	85.32 MB	24	4819	2397
Transient JMS	672 KB	3.43 MB	40.00 MB	25	2268	1121
Transient JMS	1 MB	3.44 MB	24.95 MB	26	1561	767
Persistent JMS	320 KB	3.43 MB	88.54 MB	25	4875	2425
Persistent JMS	672 KB	3.44 MB	39.90 MB	25	2268	1122
Persistent JMS	1 MB	3.46 MB	26.62 MB	27	1561	767

Table 6: DST maximum connection test result

The results of this test show that each JMS DST used two threads to handle each connection, whereas the Jini DST used one thread for each connection. Therefore given the same stack size, the same heap settings and the same free heap memory, a Jini DST is more structural scalable than a JMS DST as it can support approximately twice the *maximum DST connections* supported by the JMS DST.

Also, based on our assumption regarding the relation between the connections and the DST memory and thread usages, it is calculated that in this tests both JMS DSTs used approximately 20 KB heap memory for each thread regardless of the stack size, and the Jini DST used approximately 30 KB heap memory for each thread regardless of the stack size, and that the overall memory occupied by both heap and stacks was around 1.7 GB to 1.8 GB. Therefore it is estimated that in this test environment and given the default 320 KB JVM stack size, if all the memory is used for connections,

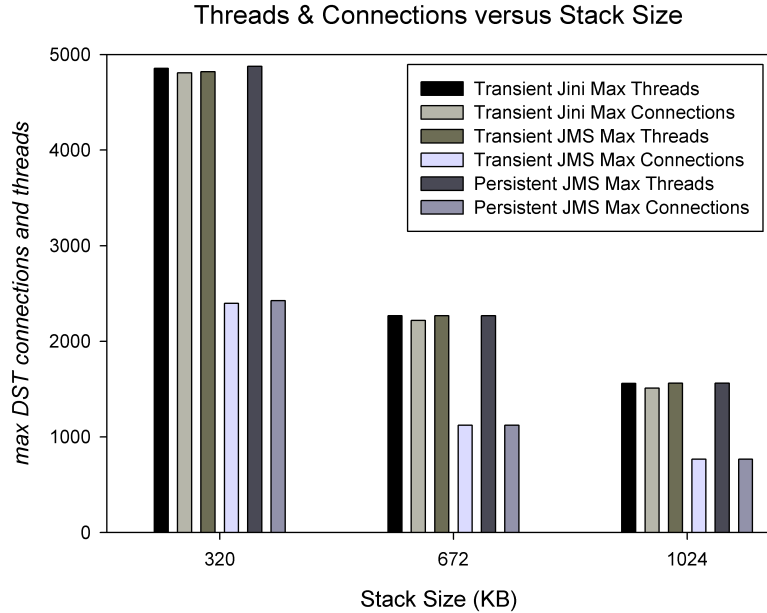


Figure 42: DST connections and threads versus stack sizes

each DST may support approximately 5,000 threads, which leads to 2,500 connections for a JMS DST, and 5,000 connections for a Jini DST.

However, when considering both the structural scalability and the space scalability, the *maximum DST connections* supported by the transient Jini DST or the transient JMS DST are expected to be fewer than their estimated amount because the demands stored in the DST also take up heap memory, and to deal with simultaneous desire for both kinds of scalability, their heap memory size needs to be increased and/or their thread stack size needs to be reduced to support more threads for DST connections. In contrast, since the persistent JMS DST can store demands in hard disk, its space scalability is less dependent on its heap memory and therefore can have less affected structural scalability, which is supported by the result of a simple test that the persistent JMS DST with 256 MB heap easily stored 10,000 32-KB payload demands while simultaneously supporting 2,400 DWTs.

3.4.3 Summary

In this section the structural scalability of the GIPSY runtime system was investigated by testing how many node/tier registrations the GMT can hold and how many

connections each DST can support. It was discovered that for a GMT with the default 64 MB heap size and when all the memory is for storing node/tier registrations, there could be approximately 40,000 node/tier registrations. However in reality, the number of node/tier registrations that the GMT can store depends on the memory usage of the GIPSY node process where the GMT is allocated. For example, when there are no frequent managerial issues arise in the GIPSY runtime system, system demands sent to the GMT are rare and the memory of the GMT used for management is therefore limited so that most of its memory can be used to store node/tier registrations; however, where managerial issues such as DST crashes or tier allocations are frequent, system demands sent to the GMT will consume much memory, and if the JVM of the GMT cannot garbage collect its heap in time, eventually the GMT will have no space for additional node/tier registrations. In the future by using a more scalable and less memory dependent data storage technology to store node/tier registrations, for example, a database, the GIPSY runtime system can easily grow much larger to have for example millions of node/tier registrations.

As to the structural scalability concerning the DSTs, when all the memory is used for DST connections, the Jini DST is more structural scalable than the JMS DSTs since it can support approximately twice the *maximum DST connections* than the JMS DSTs; whereas when there are significant requirements in the space scalability in addition to the requirements in the structural scalability, without additional memory relevant settings both the transient Jini DST and the transient JMS DST will have to trade off between the space scalability and the structural scalability due to the resource contention in their heap memory, whereas the persistent JMS DST supports both kinds of scalability well since it is able to store demands beyond its memory limit so that it is less constraint by its heap memory size.

3.5 Load Scalability

The load scalability of GIPSY runtime system concerns if the maximum *throughput* of demand processing is able to increase proportionally with the number of the tiers

that process the demands, since the GIPSY runtime system was designed to be able to allocate more tiers to handle increasing workload. To assess the load scalability of the GIPSY runtime system, the *throughput* of a π calculation demand is used, and its relation with the *number of DWTs* that perform the π calculation is studied. The π calculation demand requires a DWT to compute up to 1000 decimal places, so that the execution time of such a π calculation demand is longer than the response time of each DST observed in Figure 34, making the demand worthy of being processed remotely while generating a significant *throughput*.

Before testing the load scalability of the GIPSY runtime system, a test in which all the GIPSY tiers are allocated in the same GIPSY node shall be performed first, and the result of this test is used as the benchmark to compare with the scenario when the DWTs are allocated in multiple GIPSY nodes. After the benchmark test, the minimum number of tiers to be allocated within a single GIPSY node to sufficiently utilize its computation power needs to be determined experimentally. With this minimum number of tiers deployed in each GIPSY node, more tiers can be allocated in more GIPSY nodes and how the maximum *throughput* changes with the overall number of tiers allocated in all the GIPSY nodes is of our interest.

Since it was observed in Section 3.3 that in the sense of demand response time, the transient Jini DST is faster than the transient JMS DST that is even faster than the persistent JMS DST, the tests in this section compare the load scalability of the transient Jini DST and the transient JMS DST since they are the fastest DSTs among their kinds.

3.5.1 Load Scalability Test: All the Tiers in One GIPSY Node

Objective

To investigate how the maximum *throughput* of the GIPSY runtime system changes with the number of DWTs allocated, and in this test all the GIPSY tiers are allocated in the same GIPSY node. The result of this test is served as the benchmark in the

test in Section 3.5.3.

Scenario

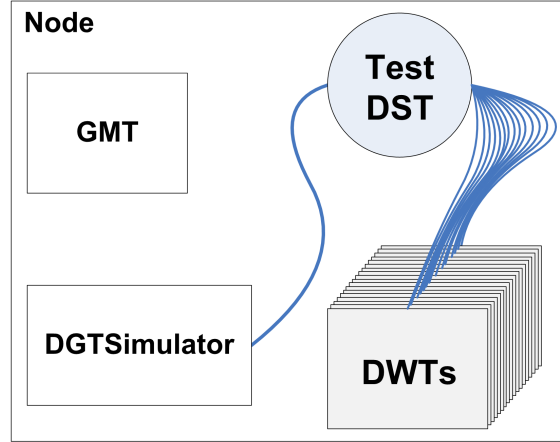


Figure 43: Load scalability test deployment: all the tiers in one GIPSY node

A GIPSY instance consisting of a single GIPSY node (computer) is used to perform this test as illustrated in Figure 43: and all the GIPSY tiers are allocated in this single GIPSY node. For each number of DWTs (beginning from 1, 2, 3, . . . , until 16 DWTs) allocated in the node, we adjust the DGT simulator threads to find the maximum *throughput* of demand computation. The relation between the maximum *throughput* and the *number of DWTs* is recorded.

Environment and Parameters

The hardware and operating system environment of the GIPSY node is shown in Table 2. The test DST is started using the profiles that were numbered 9 and 10 in Table 3 to start a transient Jini DST and a transient JMS DST respectively.

Results

Figure 44 shows that in the case when all the GIPSY tiers are allocated in the same GIPSY node, for the Jini DST, its throughput reached its approximate maximum when there were more than two DWTs allocated; however for the JMS DST,

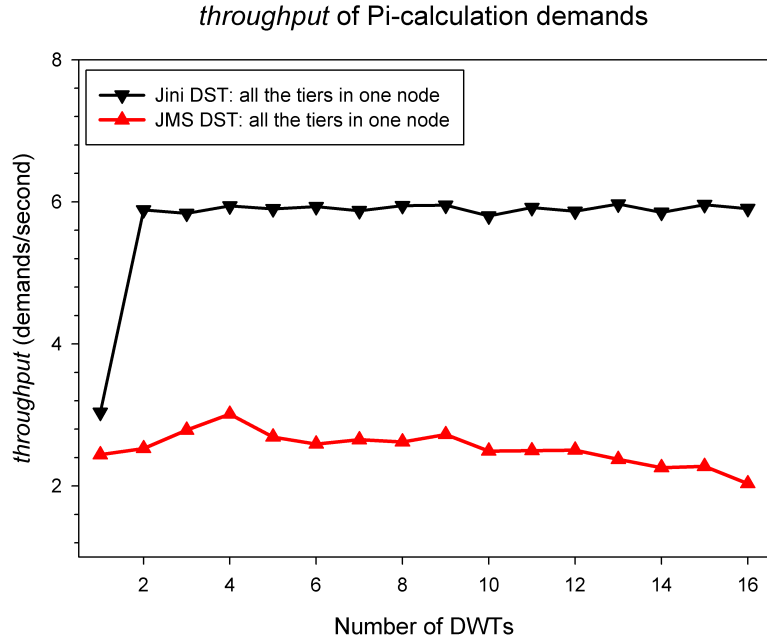


Figure 44: Load scalability test result: all the tiers in one node

its throughput increased from the beginning but when there were more than 4 DWTs allocated, its throughput gradually decreased. We attribute the throughput saturation of the Jini DST to the fact that the computer used for testing had two processors. Also we attribute the throughput decrease of the JMS DST to the resource contention due to the increasing amount of threads running in the computer. Further discussion of this test result is presented in Section 3.5.3.

3.5.2 Load Scalability Test: DWTs Scaled Out in One GIPSY Node

Objective

To find out the minimum number of DWTs to be allocated in a single GIPSY node by observing how the *throughput* of the GIPSY runtime system improves by allocating more DWTs within the single GIPSY node. The minimum number of the DWTs that can sufficiently utilize the computation power of the GIPSY node will be used in the next test in Section 3.5.3.

Scenario

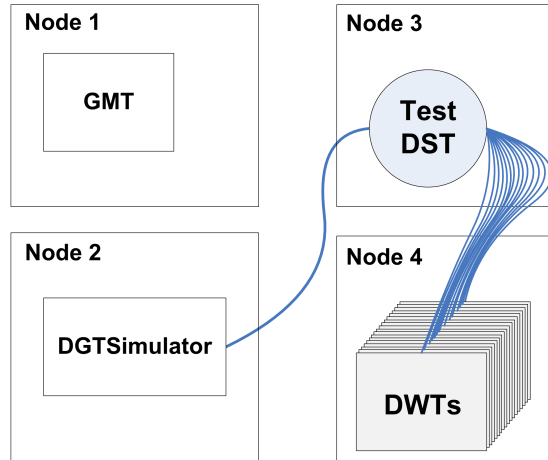


Figure 45: Load scalability test deployment: DWTs scaled out in one GIPSY node

A GIPSY instance consisting of four GIPSY nodes (computers) is used to perform this test as illustrated in Figure 45: Node 1 runs the GMT to control tier allocations, Node 2 runs the DGT simulator, Node 3 runs the test DST and Node 4 runs multiple DWTs. For each number of DWTs (beginning from 1, 2, 3, . . . , until 16 DWTs) allocated in Node 4, we adjust the DGT simulator threads to find the maximum *throughput* of demand computation. The relation between the maximum *throughput* and the *number of DWTs* allocated in Node 4 is recorded.

Environment and Parameters

The hardware and operating system environment of each GIPSY node is shown in Table 2. The test DST is started using the profiles that were numbered 9 and 10 in Table 3 to start a transient Jini DST and a transient JMS DST respectively.

Results

Figure 46 shows that for both the Jini and the JMS DSTs, their *throughput* achieved their approximate maximum when there were two DWTs allocated in the GIPSY node. This is reasonable because each computer used as the GIPSY node has two

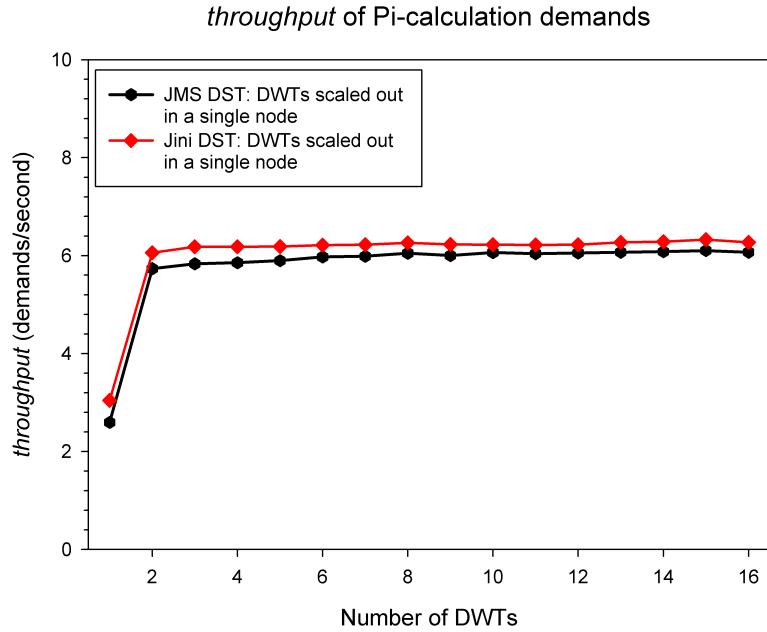


Figure 46: Load scalability test result: DSTs scaled out in one GIPSY node

processors as shown in the test environment in Table 2. Therefore it is reasonable to allocate one DWT per processor in the next test in Section 3.5.3 to make the number of DWTs proportional to the number of processors used for demand computation.

3.5.3 Load Scalability Test: DWTs Scaled Out in Multiple GIPSY Nodes

Objective

Assess the load scalability of the GIPSY runtime system by observing the relation between the maximum *throughput* provided by the GIPSY runtime system and the total number of DWTs allocated in the runtime system.

Scenario

A GIPSY instance consisting of 11 GIPSY nodes (computers) is used to perform this test as illustrated in Figure 47: Node 1 runs the GMT to manage all the GIPSY nodes, Node 2 runs the DGT simulator, Node 3 runs the test DST and all the other nodes are used to allocate DWTs, with each node allocating maximally two DWTs, i.e. one

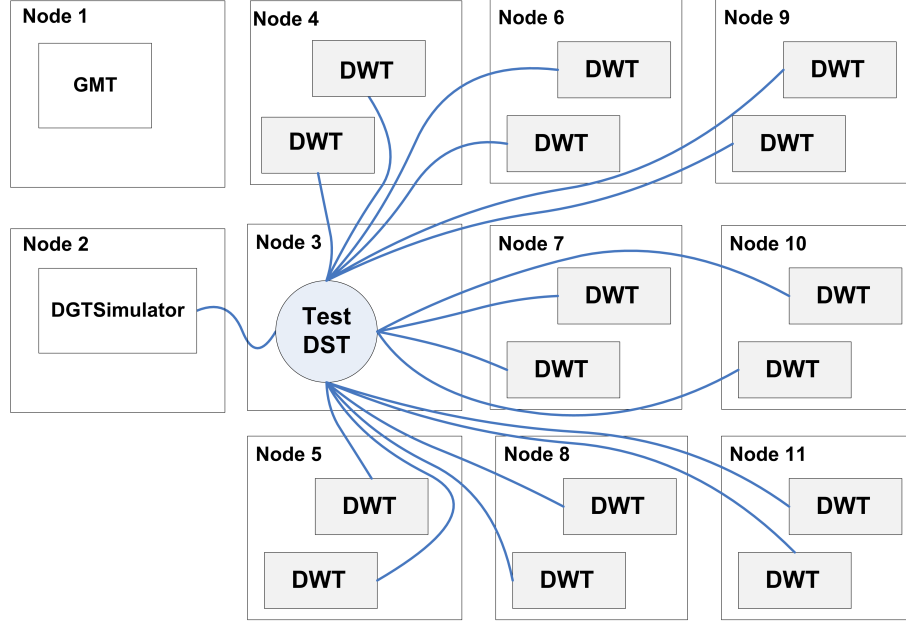


Figure 47: Load scalability test deployment: DWTs scaled out in multiple GIPSY nodes

DWT per processor. For each number of DWTs (beginning from 1, 2, 3, . . . , until 16 DWTs) allocated in the runtime system, we adjust the DGT simulator threads to find the maximum *throughput* of demand computation. The relation between the maximum *throughput* and the *number of DWTs* allocated in the entire runtime system is recorded. .

Environment and Parameters

The hardware and operating system environment of each GIPSY node is shown in Table 2. The test DST is started using the profiles that were numbered 9 and 10 in Table 3 to start a transient Jini DST and a transient JMS DST respectively.

Results

Figure 48 shows that when there were up to 16 DWTs allocated in 8 GIPSY nodes with one DWT per processor, the *throughput* provided by the Jini DST was approximately linearly increasing with the *number of DWTs* allocated, whereas the *throughput* of the JMS DST increased sub-linearly with the *number of DWTs* allocated and was almost

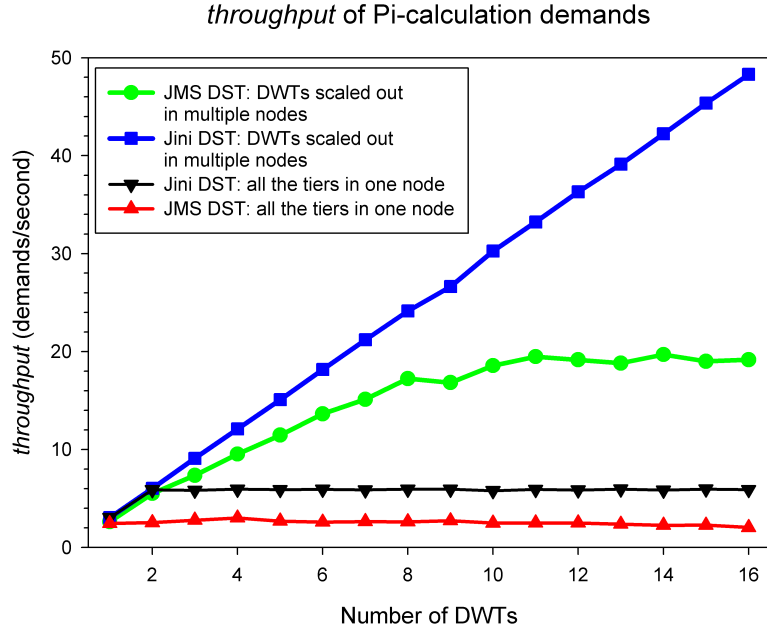


Figure 48: Load scalability test result: DSTs scaled out in multiple GIPSY nodes

flat after more than 10 DWTs were allocated. This result leads to a conclusion that in this test scenario the Jini DST is load scalable since its *throughput* increased linearly with the *number of DWTs* allocated in the GIPSY runtime system.

Also the test result in Section 3.5.1 is also presented in Figure 48 to compare the results of these two tests directly. The comparison between these two results shows that the scaling-out of the DWTs into multiple computers indeed has advantage in the overall *throughput* of the GIPSY runtime system.

However, the result of this test cannot conclude that the JMS DST is not load scalable, because its *throughput* saturation might be caused by either the possible bottleneck in the Node 2 in Figure 47 where all the DGT threads launched by the DGT simulator were running, or caused by the possible bottleneck in the Node 3 in Figure 47 where the JMS DST was running. The other GIPSY nodes in Figure 47 where the DWTs were running did not cause the *throughput* saturation, because if the *throughput* saturation was caused by the bottleneck in these DWT nodes, it should have appeared already when there were two DWTs allocated for the first time, rather than when there were 10 DWTs allocated as shown in Figure 48. To investigate if the *throughput* saturation of the JMS DST was caused by the possible bottleneck

in the GIPSY node where all the DGT threads were running, the following test was performed.

3.5.4 Load Scalability Test: Two DGT Nodes for the JMS DST

Objective

To find out if the *throughput* saturation of the JMS DST shown in Figure 48 was caused by the possible bottleneck in the GIPSY node where all the DGT threads were running.

Scenario

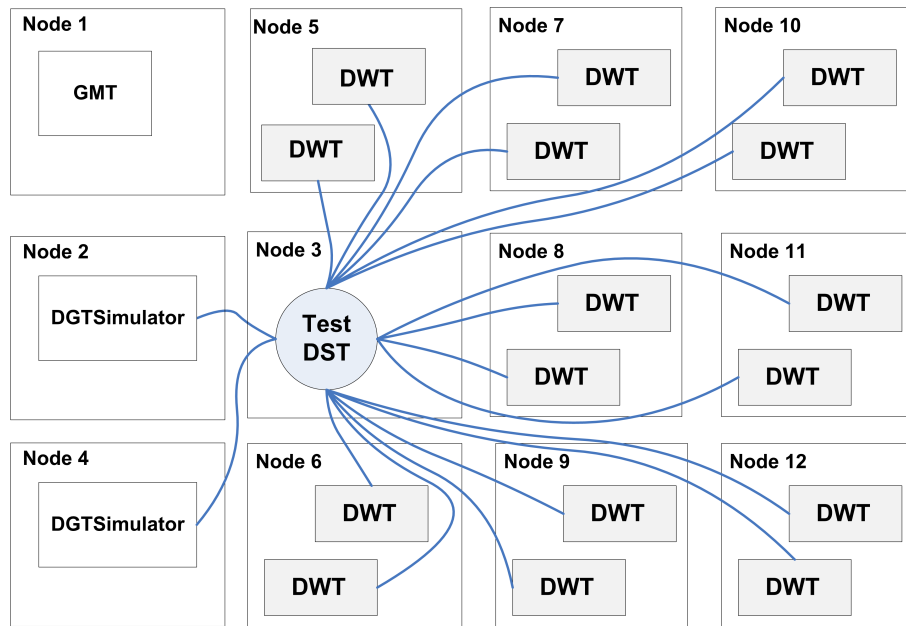


Figure 49: Load scalability test deployment: two DGT nodes for the JMS DST

A GIPSY instance consisting of 12 GIPSY nodes (computers) is used to perform this test as illustrated in Figure 49: Node 1 runs the GMT to manage all the GIPSY nodes, Node 2 runs the DGT simulator, Node 3 runs the JMS DST, Node 4 runs the second DGT simulator, and the remaining nodes are used to allocate DWTs

with two DWTs per node, i.e. one DWT per processor. For each number of DWTs (beginning from 2, 4, 6, . . . , until 16 DWTs) allocated, use the 2 DGT simulators allocated in the two different GIPSY nodes to achieve the maximum *throughput* of demand computation. Since in this test there are 2 DGT simulators running in two different GIPSY nodes, compared to the previous test when only one DGT simulator was used, each one of the two DGT simulators in this test only launches half the number of the DGT threads as those launched in the previous test. In this way if the *throughput* saturation of the JMS DST in the previous test was indeed caused by the bottleneck in the GIPSY node where the DGT threads were running, then in this test the *throughput* saturation should appear much later because in this test each GIPSY node now has fewer (only half of) DGT threads running so that the possible bottleneck is reduced and postponed; however if the *throughput* saturation in this test still appears at the same time when there are 10 DWTs allocated as it did in the previous test as shown in Figure 3.5.3, then the *throughput* saturation in the previous test was not caused by the bottleneck in DGT threads, which implies that the *throughput* saturation in the previous test was caused by the JMS DST itself.

Environment and Parameters

The hardware and operating system environment of each GIPSY node is shown in Table 2. The test DST running in Node 3 is started using DST profile 10 to start the transient JMS DST.

Results

Figure 50 shows that the *throughput* curve of the JMS DST in this test was similar to the *throughput* curve of the JMS DST in the previous test. This result leads to the deduction that the *throughput* saturation of the JMS DST in the test in Section 3.5.3 as shown in Figure 48 was caused by the JMS DST itself rather than the DGT threads, because otherwise the *throughput* saturation in this test should have appeared later since in this test the DGT threads were distributed into two GIPSY nodes to halve the resource contention in each DGT node. Based on the result of this test and the

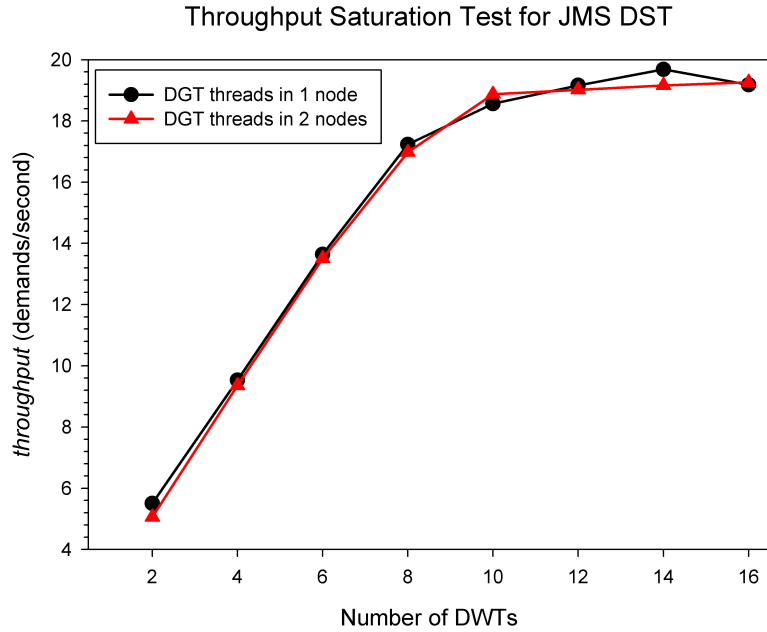


Figure 50: Load scalability test result: two DGT nodes for the JMS DST

previous test in Section 3.5.3, it is concluded that the Jini DST is more load scalable than the JMS DST, because the *throughput* of the Jini DST increased linearly with the *number of DWTs* allocated in the GIPSY runtime system, whereas *throughput* of the JMS DST saturated when more than 10 DWTs were allocated, indicating that the JMS DST could not utilize the computation resources as well as the Jini DST did in handling increasing workload.

3.5.5 Summary

In this section the load scalability of GIPSY runtime system was investigated by studying and comparing the relation between the *throughput* provided by each DST and the *number of DWTs* allocated in the GIPSY runtime system. It is concluded that the Jini DST is more load scalable than the JMS DST as in the test scenarios the maximum *throughput* of the Jini DST increased almost linearly with the *number of DWTs* allocated in the runtime system, whereas the *throughput* of the JMS DST reached its saturation when there were more than 10 DWTs allocated.

3.6 Conclusion

In this chapter the four kinds of scalability of GIPSY runtime were assessed via different experiments. In the GIPSY runtime system, the persistent JMS DST provides the best space scalability since it can store demands in its persistent storage beyond its memory limit, whereas the transient Jini DST provides the best space-time scalability since its demand sending and reading speeds are not undermined by the amount of demands it stores before its memory goes critical. The transient Jini DST also provides the best structural scalability since it uses one thread to handle each connection, whereas a JMS DST uses two threads for each connection; and it offers the best load scalability as well since its *throughput* of demand migration and process increases linearly with the increase in the *number of DWTs* allocated in the GIPSY runtime system. However, when requirements in all the four kinds of scalability exist and the requirement in space scalability dominates, the persistent JMS DST survives longer than other DSTs as it can store demands into its persistent storage to release its memory for other requirements in such as the number of DGT/DWT connections. Therefore for demands that require little demand storage space but fast response speed, the transient Jini DST is recommend; whereas for demands that require large demand storage space but are not very response-time sensitive, the persistent JMS is recommended. By using all the DSTs wisely in a mixed fashion and by expanding the system via node registration and tier allocation, the GIPSY runtime system can easily deal with increasing demand processing and storage requirements, and thus is concluded to have good scalability.

Chapter 4

Conclusions and Future Work

4.1 Conclusions

The work presented in this thesis developed the GIPSY runtime system using the multi-tier architecture and assessed the scalability of the GIPSY runtime system developed. More specifically this thesis:

- presented the development of a configuration system for the GIPSY runtime to achieve flexibility for adding new tier implementations without changing the source code of the existing system components affected.
- presented the design and the implementation of the DSTs, the GMT and the GIPSY node for the GIPSY instance bootstrap process, the GIPSY node registration process, the GIPSY tier allocation process and the GIPSY tier deallocation process, so that the GIPSY runtime system can be easily expanded over multiple computers in a network.
- researched the scalability evaluation methodologies to find the four types of scalability applicable to the GIPSY runtime system, i.e. the space scalability, the space-time scalability, the structural scalability and the load scalability, as well as their metrics.
- presented the assessment of the four types of scalability of the GIPSY runtime

system via various tests.

The scalability assessment shows that the persistent JMS DST provides the best space scalability and is suitable for demands requiring large storage space with relatively slower migration speed. Also the transient Jini DST provides the best space-time scalability, structural scalability and the best load scalability, and is suitable for demands requiring fast migration speed with relatively smaller storage space requirements. As demonstrated in this thesis, the GIPSY runtime system is able to deal with increasing workload and demand storage requirements by registering more computers as the GIPSY nodes and allocating more GIPSY tiers in the registered GIPSY nodes, therefore the GIPSY runtime system is scalable.

4.2 Limitations and Future Work

Limitations regarding the four types of scalability exist in the GIPSY runtime system developed in this thesis, and shall be resolved in the future work. Specifically:

- For the DSTs whose space scalability is limited by their memory, including the transient Jini DST, the transient JMS DST and the persistent Jini DST, to resolve this limitation, one could allocate more DSTs in the GIPSY runtime system so that although the space scalability of a single DST is still constrained by its memory, the entire GIPSY runtime system can store increasing amount of demands beyond the memory limit of the single DST. However, this solution is constrained by the total memory available in the GIPSY runtime system, therefore does not work in the situation that the entire system is not able to expand anymore.

Another solution is to apply a garbage collection mechanism to release the memory of the DSTs when necessary, so that none of the DSTs is likely to run out of memory in their long-term usage even if no more DSTs can be allocated in the GIPSY runtime system. It is possible to rely on the Jini DST and the JMS DST themselves to eventually release the memory taken by demands. For

example, by shortening the `Jini Lease` time, demands will be eventually deleted from `JavaSpace` in the order they were stored, and by renewing the lease when the demand is read, demands that are most frequently used may survive in the `JavaSpace`. However, `Jini` provides no mechanism to prevent memory crash in the case that a huge amount of demands are flooded into the `Jini DST` within a very short time, therefore an external flow-control and garbage collecting mechanism is necessary for the long-term use of the `Jini DST`. The situation of the `JMS DST` is different: `Sun Java Message Queue` provides options and approaches to prevent the broker from crash when its memory goes critical, for example, by dropping newest messages or deleting oldest messages [48]; however, it is unable to delete the “least frequently used” messages unless they are deleted by an external `JMS` client, therefore again an external garbage collector is required for the `JMS DST` in the case that the existing mechanisms provided by the `JMS` broker do not meet the requirements.

- For the limitation of the `JMS DSTs` in the space-time scalability, since the result-reading time provided by the `JMS DSTs` increases linearly with the amount of demands stored, the garbage collection mechanism discussed above can also be used here to remove demands from the `JMS DSTs`, so that the amount of demands stored in the `DST` can be kept under a certain level and the corresponding result-reading time can be reduced as well.
- For the limitation of the `GMT` in the structural scalability, since currently the `GMT` uses a `GMTInfoKeeper` to store all the node/tier registrations in memory, it is suggested to replace the `GMTInfoKeeper` with more scalable storage technologies such as a database, so that the growth of the system in the sense of node/tier registrations is not constrained by the memory limit of the `GMTInfoKeeper`.
- For the limitation of the `JMS DST` in the load scalability, one could allocate more `JMS DSTs` in the `GIPSY` runtime system to distribute the load of the demand migration and processing among all the allocated `DSTs`, so that the

overall throughput of the demand migration and processing provided by the entire GIPSY runtime system is not constrained by the throughput saturation of a single JMS DST.

Besides, the scalability tests presented in this thesis have limits in how the system was scaled, since the GIPSY runtime system was tested in the computers running 32-bit Windows 7 in the local area network in the same lab. To further investigate the scalability, the tests presented in this thesis shall be repeated in other environments, such as in computers with different network connectivity and/or running Linux or Mac OS, or in clusters or computers with high number of processors. Also, in the further the load scalability of the runtime system shall be tested with DWTs allocated in more computers to investigate when the throughput provided by the Jini DST will saturate. The number of DWTs corresponding to the beginning of the throughput saturation is referred as throughput saturation point. With the throughput saturation point discovered, the system can be further scaled out by allocating more DSTs connected to more DWTs, with the number of DWTs connected to each single DST below its corresponding throughput saturation point.

Bibliography

- [1] J. Paquet, “Distributed eductive execution of hybrid intensional programs,” in *Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC’09)*, (Seattle, Washington, USA), pp. 218–224, IEEE Computer Society, July 2009.
- [2] J. Paquet, *Scientific Intensional Programming*. PhD thesis, Department of Computer Science, Laval University, Sainte-Foy, Canada, 1999.
- [3] W. Du and W. W. Wadge, “A 3D spreadsheet based on intensional logic,” *IEEE Software*, vol. 7, pp. 78–89, June 1990.
- [4] W. Du and W. W. Wadge, “The eductive implementation of a three-dimensional spreadsheet,” *Software Practice and Experience*, vol. 20, pp. 1097–1114, Nov. 1990.
- [5] W. Du, *Indexical Parallel Programming*. PhD thesis, Department of Computer Science, Victoria University, Canada, 1991.
- [6] A. A. Faustini and E. B. Lewis, *Towards a Real-Time Dataflow Language*. Los Alamitos, CA, USA: IEEE Computer Society, 1989.
- [7] J. Plaice, R. Khedri, and R. Lalement, “From abstract time to real time,” in *In Proceedings of the Sixth International Symposium on Lucid and Intensional Programming*, pp. 83–93, 1993.
- [8] J. Plaice and W. W. Wadge, “A new approach to version control,” *IEEE Transactions on Software*, vol. 19, pp. 268–276, Mar. 1993.

- [9] W. W. Wadge, G. Brown, M. C. Schraefel, and T. Yildirim, “Intensional HTML,” in *4th International Workshop PODDP’98*, Mar. 1998.
- [10] J. Paquet and J. Plaice, “On the design of an indexical query language,” in *Proceedings of the Seventh International Symposium on Lucid and Intensional Programming*, pp. 28–36, 1994.
- [11] W. W. Wadge and E. A. Ashcroft, *Lucid, the Dataflow Programming Language*. London: Academic Press, 1985.
- [12] E. A. Ashcroft and W. W. Wadge, “Lucid – a formal system for writing and proving programs,” *SIAM J. Comput.*, vol. 5, no. 3, 1976.
- [13] E. A. Ashcroft and W. W. Wadge, “Erratum: Lucid – a formal system for writing and proving programs,” *SIAM J. Comput.*, vol. 6, no. 1, p. 200, 1977.
- [14] E. A. Ashcroft and W. W. Wadge, “Lucid, a nonprocedural language with iteration,” *Communications of the ACM*, vol. 20, pp. 519–526, July 1977.
- [15] E. A. Ashcroft, A. A. Faustini, R. Jagannathan, and W. W. Wadge, *Multidimensional Programming*. London: Oxford University Press, Feb. 1995. ISBN: 978-0195075977.
- [16] R. Jagannathan and C. Dodd, “GLU programmer’s guide,” tech. rep., SRI International, Menlo Park, California, 1996.
- [17] R. Jagannathan, C. Dodd, and I. Agi, “GLU: A high-level system for granular data-parallel programming,” in *Concurrency: Practice and Experience*, vol. 1, pp. 63–83, 1997.
- [18] 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, Oct. 2005. ISBN 0494102934; online at <http://arxiv.org/abs/0907.2640>.

- [19] P. Grogono, S. Mokhov, and J. Paquet, “Towards JLucid, Lucid with embedded Java functions in the GIPSY,” in *Proceedings of the 2005 International Conference on Programming Languages and Compilers (PLC 2005)*, pp. 15–21, CSREA Press, June 2005.
- [20] A. Wu, J. Paquet, and S. A. Mokhov, “Object-Oriented Intensional Programming: A New Concept in Object-Oriented and Intensional Programming Domains.” Unpublished, 2007.
- [21] A. Wu, J. Paquet, and S. A. Mokhov, “Object-Oriented Intensional Programming: Intensional Classes Using Java and Lucid.” Unpublished, 2008.
- [22] A. Wu, J. Paquet, and S. A. Mokhov, “Object-oriented intensional programming: Intensional Java/Lucid classes,” in *Proceedings of SERA 2010*, pp. 158–167, IEEE Computer Society, 2010. Online at: <http://arxiv.org/abs/0909.0764>.
- [23] W. W. Wadge, “Hamming’s problem example.” [online], Dec. 2003. <http://i.csc.uvic.ca/home/hei/lup/contents.html>.
- [24] A. H. Wu, “Semantic checking and translation in the GIPSY,” Master’s thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2002.
- [25] J. Paquet, P. Grogono, and A. H. Wu, “Towards a framework for the general intensional programming compiler in the GIPSY,” in *Proceedings of the 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004)*, (Vancouver, Canada), ACM, Oct. 2004.
- [26] J. Paquet and P. Kropf, “The GIPSY architecture,” in *Proceedings of Distributed Computing on the Web*, (Quebec City, Canada), 2000.
- [27] B. Lu, *Developing the Distributed Component of a Framework for Processing Intensional Programming Languages*. PhD thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, Mar. 2004.

- [28] L. Tao, “Warehouse and garbage collection in the GIPSY environment,” Master’s thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2004.
- [29] A. H. Pourteymour, E. Vassev, and J. Paquet, “Towards a new demand-driven message-oriented middleware in GIPSY,” in *Proceedings of PDPTA 2007*, (Las Vegas, USA), pp. 91–97, PDPTA, CSREA Press, June 2007.
- [30] Jini Community, “Jini network technology.” [online], Sept. 2007. <http://java.sun.com/developer/products/jini/index.jsp>.
- [31] 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.
- [32] Sun Microsystems, Inc., “Java Message Service (JMS).” [online], Sept. 2007. <http://java.sun.com/products/jms/>.
- [33] A. H. Pouteymour, “Comparative study of Demand Migration Framework implementation using JMS and Jini,” Master’s thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, Sept. 2008.
- [34] B. Han, S. A. Mokhov, and J. Paquet, “Advances in the design and implementation of a multi-tier architecture in the GIPSY environment with Java,” in *Proceedings of SERA 2010*, pp. 259–266, IEEE Computer Society, 2010. Online at <http://arxiv.org/abs/0906.4837>.
- [35] B. Han, “Towards a multi-tier runtime system for GIPSY,” Master’s thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2010.

- [36] L. Duboc, D. S. Rosenblum, and T. Wicks, “A framework for characterization and analysis of software system scalability,” in *ESEC/SIGSOFT FSE* (I. Crnkovic and A. Bertolino, eds.), pp. 375–384, ACM, Sept. 2007.
- [37] A. B. Bondi, “Characteristics of scalability and their impact on performance,” in *Proceedings of the 2nd international workshop on Software and performance*, pp. 195–203, 2000.
- [38] T. Agerwala and S. Chatterjee, “Computer architecture: Challenges and opportunities for the next decade,” *IEEE Micro*, vol. 25, no. 3, pp. 58–69, 2005.
- [39] M. M. Michael, J. E. Moreira, D. Shiloach, and R. W. Wisniewski, “Scale-up x scale-out: A case study using Nutch/Lucene,” in *Proceedings of IPDPS*, pp. 1–8, IEEE, 2007.
- [40] P. Dube, H. Yu, L. Zhang, and J. E. Moreira, “Performance evaluation of a commercial application, trade, in scale-out environments,” in *MASCOTS*, pp. 252–259, IEEE Computer Society, 2007.
- [41] M. D. Hill, “What is scalability?,” *SIGARCH Comput. Archit. News*, vol. 18, no. 4, pp. 18–21, 1990.
- [42] D. Jin and S. G. Ziavras, “Robust scalability analysis and SPM case studies,” *The Journal of Supercomputing*, vol. 43, no. 3, pp. 199–223, 2008.
- [43] P. Jogalekar and C. M. Woodside, “Evaluating the scalability of distributed systems,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 11, no. 6, pp. 589–603, 2000.
- [44] Y. Chen, X.-H. Sun, and M. Wu, “Algorithm-system scalability of heterogeneous computing,” *J. Parallel Distrib. Comput.*, vol. 68, no. 11, pp. 1403–1412, 2008.
- [45] E. Cecchet, J. Marguerite, and W. Zwaenepoel, “Performance and scalability of EJB applications,” in *OOPSLA*, pp. 246–261, 2002.
- [46] D. Green, “Java reflection API.” Sun Microsystems, Inc., 2001–2005. <http://java.sun.com/docs/books/tutorial/reflect/index.html>.

- [47] Apache River Community, “Apache River.” [online], 2010. <http://incubator.apache.org/river/index.html>.
- [48] Sun Microsystems, Inc., *Sun Java System Message Queue 4.3 Administration Guide*. 4150 Network Circle, Santa Clara, California 95054, U.S.A.: Sun Microsystems, Inc., 2008.
- [49] Sun Microsystems, Inc., *Memory Management in the Java HotSpot™ Virtual Machine*. 4150 Network Circle, Santa Clara, California 95054, U.S.A.: Sun Microsystems, Inc., 2006.
- [50] Sun Microsystems, Inc., *Java SE Monitoring and Management Guide*. 4150 Network Circle, Santa Clara, California 95054, U.S.A.: Sun Microsystems, Inc., 2006.
- [51] MSDN, “Virtual address space.” [online], Dec. 2010. [http://msdn.microsoft.com/en-us/library/aa366912\(v=VS.85\).aspx](http://msdn.microsoft.com/en-us/library/aa366912(v=VS.85).aspx), last viewed January 2011.
- [52] Sun Microsystems, Inc., *Sun Java System Application Server Enterprise Edition 8.1 Performance Tuning Guide*. 4150 Network Circle, Santa Clara, California 95054, U.S.A.: Sun Microsystems, Inc., 2005.
- [53] Oracle, “Frequently asked questions about the Java HotSpot VM.” [online], 2005. <http://www.oracle.com/technetwork/java/hotspotfaq-138619.html>, last viewed January 2011.
- [54] MSDN, “Thread stack size.” [online], Dec. 2010. [http://msdn.microsoft.com/en-us/library/ms686774\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/ms686774(v=vs.85).aspx), last viewed January 2011.