NOTE TO USERS

GSHELL: A COMMAND INTERPRETER FOR A PUBLIC COMPUTING UTILITY

by

Ying Deng

November 2004

School of Computer Science
McGill University
Montréal, Canada

A thesis submitted to McGill University
in partial fulfillment of the requirements of
the degree of Master of Science

Your file Votre référence
ISBN: 0-494-12431-8
Our file Notre référence
ISBN: 0-494-12431-8

ABSTRACT

Galaxy shell, a sub component of Galaxy project is designed and implemented. Galaxy shell is a core component of the Galaxy services, which provides an interface for Galaxy developers or other modules to access certain functions supported by Galaxy system. Through Galaxy shell, developers can search, query, and bind resources according to static or dynamic constraints, launch commands either locally or remotely. Galaxy shell also supports some special functions, such as remote pipeline and multicast remote execution. Galaxy shell not only provides a valuable tool for the research and development of Galaxy project, but also can be integrated into the core management module of Galaxy, Distributed Resource Allocation Manager to realize automatic discovery, dissemination, and allocation Galaxy resources. The specification of Galaxy shell is also provided.

Galaxy shell is implemented based on Project JShell, an emulation of a UNIX shell written in Java. Galaxy shell extends JShell by adding new commands special for Galaxy resource management and consumption. Project JavaCC, a parser generator and lexical analyzer generator is used for writing the token manager and parser for Galaxy shell. Apache XML-RPC is used to implement remote procedure calls.

RESUMÉ

La coquille de Galaxy, un sous composant du projet Galaxy est conçue et mise en application. La coquille de Galaxy est un composant de noyau des services Galaxy qui fournit une interface pour des développeurs de Galaxy ou autres modules pour accéder à certaines fonctions soutenues par le système Galaxy. Par la coquille de Galaxy, les développeurs peuvent rechercher, questionner et lier des ressources en tenant compte des contraintes statiques et dynamiques, lancer les commandes autant localement qu'à distance. La coquille de Galaxy soutient également quelques fonctions spéciales telles que les chemins d'accès à distance et l'exécution à distance de plusieurs tâches. La coquille de Galaxy fournit non seulement un outil valable pour la recherche et le développement du projet Galaxy, mais également peut être intégrée dams le noyau du module de gestion de Galaxy, du gestionnaire attribué d'allocation des ressources pour la réalisation de découverte automatique, de la diffusion et de l'allocation des ressources de Galaxy. Les spécifications de la coquille de Galaxy sont également fournies.

La coquille de Galaxy est mise en application basée sur le projet JShell, une émulation d'une coquille d'UNIX écrite en JAVA. La coquille prolonge JShell pour en ajoutant de nouvelles commandes spéciales pour la gestion des ressources Galaxy et pour la consommation. Projet JavaCC, générateur de symbolique et générateur d'analyseur lexicologique est employé pour écrire le gestionnaire de symbolique et analyseur pour la coquille de Galaxy. XML-RPC Apache est emploé pour mettre en application des appels à distance de procédé.

ACKNOWLEDGEMENTS

The author sincerely thanks Professor Muthucumaru Maheswaran for his expert guidance and encouragement throughout the course of this research program. The author expresses his gratitude to Balasubramaneya Maniymaran and Shah Asaduzzaman for their great assistance in the design and implementation of the Galaxy Shell.

The research presented in this thesis was carried out in the Advanced Networking Research Lab at McGill University. The author extends thanks to Paul Card for his assistance in the laboratory. Special thanks are also extended to Leying Zhu and Beidi Chen for their assistance during this study. The assistance of the office staff in the School of Computer Science is gratefully acknowledged.

Finally, the author would like to thank his wife for her understanding and support throughout his stay at McGill University.

Ying Deng
June, 2004

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Galaxy project

Galaxy is a project that implements a public computing utility (PCU). A PCU is a generalized resource provisioning system that is composed from geographically distributed resources. The resources can be categorized into many types, such as CPU time, storage, network bandwidth, and etc. All these available resources are located at distributed domains and the clients of these resources are also distributed. There are many problems need to be resolved for the success of a PCU system, such as scalable distributed resource management, resource naming, resource discovery and dissemination, setting and disseminating administrative policies, handling faults, job scheduling, security management, and service level agreement. The design of Galaxy combines both peer-to-peer (P2P) and Grid computing ideas: a P2P overlay substrate for connecting the resources in a global network and a community-based decentralized resource management system. The P2P substrate names and locates resources as standardized virtual commodities. At the top of the P2P layer, PCU services are plugged into the P2P substrate and create a community-oriented architecture for the PCU, where services are bound to the resources in a dynamic and flexible manner. Galaxy provides a utility like interface to the virtual pool similar to that provided by electricity and water departments. By this move, it helps to neutralize the resource providers and simplifies activities such as metering and billing. The benefits and challenges of a utility computing lie on efficiently realizing the commoditization process in distributed computing systems. While Galaxy shares several ideas with other utility computing systems, it differs most of them because it is designed and implemented with:

(i) Commoditization at the core and utilizes this notion to efficiently implement resource naming and discovery

(ii) Relaxed participation models to induct public resources into the system

(iii) Geographically scalable resource management architecture

1.2 Galaxy shell project

Figure 1.1 shows a layered architecture of Galaxy. It has three layers from top which are application, middleware and resource pool. At the bottom is the resource pool composed by joined resources. In the middle is Galaxy middleware which is the core of Galaxy project and will be detailed shortly. At application layer, many applications can be built on for special tasks or organizations, such as file exchange system, storage sharing system, computing power sharing system, and etc. The middleware is composed by three levels, the bottom one is Resource Addressable Network (RAN) which provides the resource naming, discovery, and access services to the PCU. The next upper layer is Galaxy resource management system (GRMS), built on top of RAN, GRMS provides several services such as resource allocation, incentive management, and trust management to the PCU. The next upper layer of Galaxy is the service layer which includes many services such as application level QoS managers, network file system and Galaxy shell.

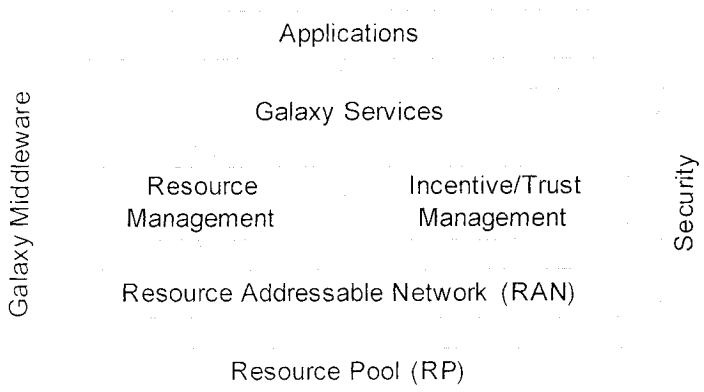

Figure 1.1 Galaxy architecture layer

(Taken from "Design of a Quality of Service Aware Public Computing Utility", Maheswaran et al., School of Computer Science, McGill University)

Galaxy shell is responsible either for sending queries to GRMS for searching certain resources, or sending request to some resources (nodes) in resource pool to remotely execute commands there (consume the public resources). From this architecture, it is notable that the shell is playing a key role in Galaxy: first, the shell can be used as a development tool for Galaxy developers to test and debug Galaxy system, select the best address methods and searching algorithm to optimize the resource management. Second, the shell can be integrated into the GRMS module, combined with

security and service level agreement to build an automatic resource allocation and management module for controlling the whole Galaxy system.

In order to communicate with the Galaxy system, each resource joined Galaxy resource pool will be wrapped into a Galaxy Daemon (GD) (See Figure 1.2). GD is a distributed sub component of the GRMS. Several GRMS functions are embedded in GD and some of these include generating status updates, sending or authenticating access requests, issuing resource usage right tokens (resource tickets), monitoring ongoing resource accesses for quality of service violations, and reporting quality of service violations.

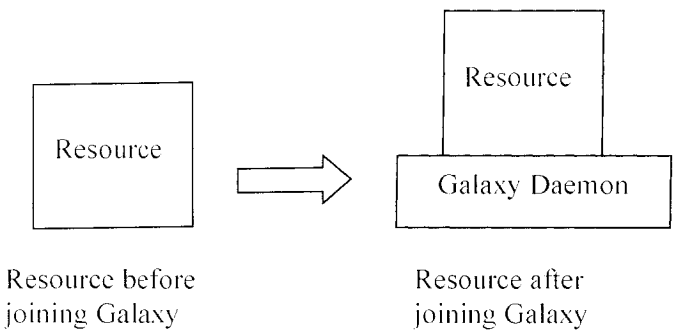

Figure 1.2: Galaxy daemon on a resource

1.2.1 Responsibilities of Galaxy shell

The main task of Galaxy shell is to providing an easy to use interface for searching, binding, allocating and consuming resources in the resource pool. It will ease the Galaxy research on GRMS which is the heart of Galaxy, GRMS. From this task, Galaxy shell needs to support following responsibilities.

1.2.1.1 Search resources

Searching for certain resources in the resource pool based on certain constraints. These constraints could be either static or dynamic. Static constraints include domain and type, domain is a set of computing resources aggregated according to certain rules and type is a special feature created and used by Galaxy GRMS to filter heterogeneous machines thus to improve efficiency

and performance in resource management. Dynamic constraint includes load, speed, lease time, and network traffic etc. Galaxy resource naming system provides support for this function. The searching result can be used for binding and executing directed execution commands at a later time.

1.2.1.2 Bind resources

The searching commands will return available resources according to the searching constraints. Before using these resources, they need to be bound with the current node (the node where the shell is running). The bind command binds current node with remote nodes through certain protocol where all sides agree, after binding, the distributed nodes can talk to each other.

1.2.1.3 Directed execution on a resource collection

After binding, following commands can be directed and executed on a given machine or machines remotely. After the command execution finished, the execution result (if any) will be sent back to the console of the shell that originally sent out the commands. The directed execution command can include remote redirection function, which means that the outcome of an execution will directed to the next machine, who is the consumer of the running result. After all the chain-like execution finished, the result (success, failure, or some other data) will be sent back the shell.

1.3 Related projects

There are many related projects were researched, both in academic and industrial area. Following is a review about these projects.

1.3.1 JXTA

Project JXTA is an open-source project originally conceived by Sun Microsystems, Inc. JXTA technology is designed to facilitate developing P2P system, which enable interconnected peers to easily locate and communicate with each other, and offer services to each other seamlessly across different platforms and networks. Project JXTA provides a framework for developing distributed computing applications and support the common functions required by any P2P system. With JXTA, formerly daunting framework-building work is waived, thus, enterprises can focus on

creating innovative software applications, and not re-inventing the "wheel". JXTA is divided into three layers (see Figure 1.3), which are platform layer (JXTA Core), services layer and applications layer.

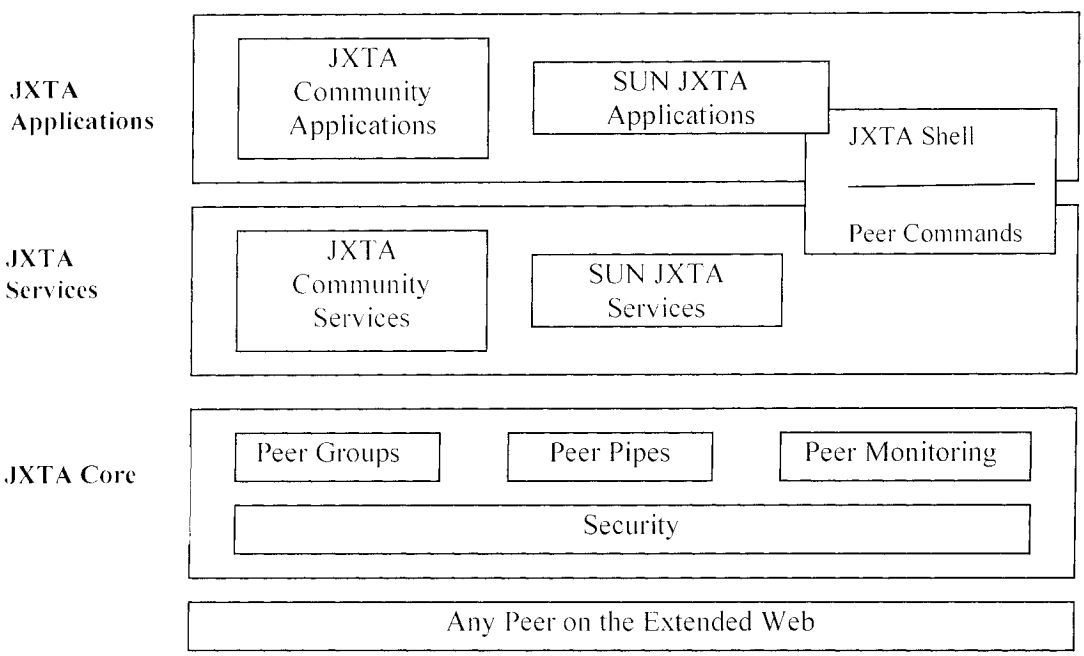

Figure 1.3: JXTA architecture

The platform layer, also known as the JXTA core, provides minimal and essential primitives that are common to P2P networking. The services layer includes optional but common and desirable network services for a P2P network including searching and indexing, directory, storage systems, file sharing, distributed file systems, resource aggregation and renting, protocol translation, authentication, and etc. The applications layer includes implementation of integrated applications which can be seen in daily life, such as file sharing, P2P Email systems, distributed auction systems, and etc.

JXTA Shell is an application build on JXTA platform; it provides interactive access to the JXTA platform via a simple command line interface, just like a UNIX shell. The difference is that the JXTA Shell is designed to be executed in a networked environment. What happens under the cover is that a user command is likely to generate a sequence of message exchanges between a set of peers, with some computation occurring on remote peer nodes, and with the answer being returned to the user.

There are many similarities between JXTA Shell and Galaxy shell: they are both targeting a distributed system; both can launch searching and binding remote nodes, and execute commands remotely. However, they do have many differences in their goal, infrastructure and functionality.

- Goal

JXTA Shell is a project written by JXTA community basically to demo the functions and values of JXTA platform, thus it is more in the application level instead of service level, although the line between the two levels is a little bit blur for JXTA Shell. JXTA Shell provides many basic "plumbing" commands such as creating pipeline, publishing advertisement, and etc. Galaxy shell aims to assist Galaxy project research, especially of RAN research, thus it doesn't provide primitive demonstrative commands like JXTA Shell, on the contrary, Galaxy shell masks these work and provides a more abstract interface for Galaxy developers.

- Infrastructure

JXTA Shell is based on a primitive resource pool, which is composed by basic peer groups; those groups are categorized randomly without any specific meaning. Galaxy shell is based on Galaxy resource pool, which is well defined and organized according to Galaxy resource naming system. This infrastructure assigns new tasks to Galaxy shell to investigate its functionality and scalability. Galaxy shell provides many new methods, which are specifically customized for this infrastructure, such as type searching, direct execution on a resource collection, etc.

- Functionality

JXTA Shell is based on a primitive resource pool, which is composed by basic peer groups; those groups are categorized randomly without any specific meaning. Galaxy shell is based on Galaxy resource pool, which is well defined and organized according to Galaxy resource naming system. This infrastructure assigns new tasks to Galaxy shell to investigate its functionality and scalability. Galaxy shell provides many new methods, which are specifically customized for this infrastructure, such as type searching, direct execution on a resource collection, etc.

1.3.2 GLUnix

GLUnix was started in 1993 as the global operating system layer for the U.C. Berkeley NOW (Network of Workstation) project, which was to construct a platform that can execute interactive parallel and sequential jobs on a cluster with negligible slowdown. A NOW is capable of hiring available cluster resources (CPU, disk, memory, network) to guarantee the performance of any workstation in the cluster. GLUnix extends some existing UNIX abstractions and introduces new abstractions, such as network programs (NPID), Parallel programs and virtual mode numbers (SPMD, VNN), signal delivery, I/O redirection (stdout, stderr), and parallel program support (barriers, coscheduling).

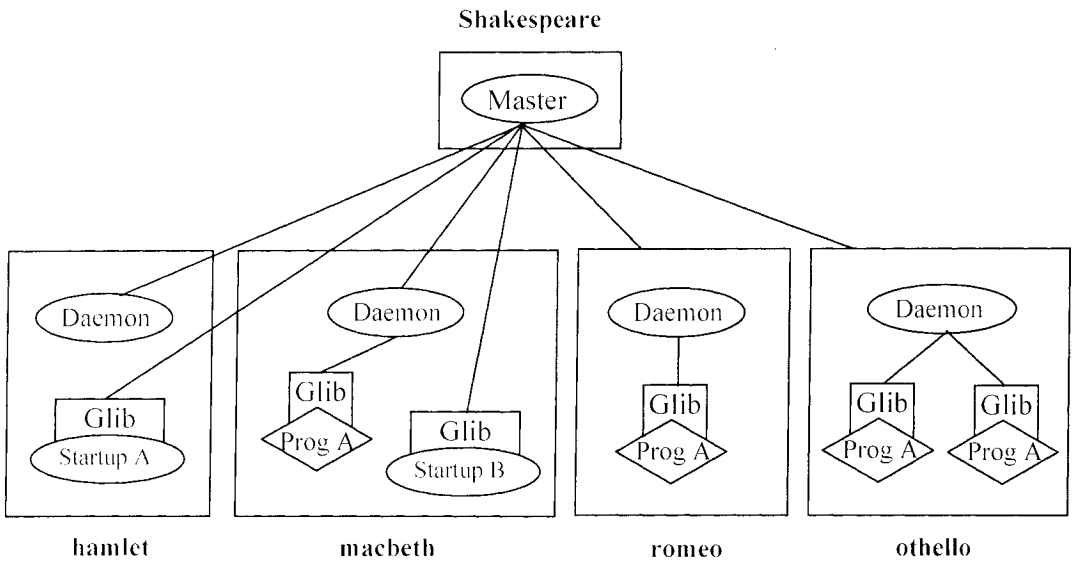

Figure 1.4: GLUnix architecture

(taken from "GLUnix: a Global Layer Unix for a Network of Workstations",
D.P. Ghormley et al. 1998 [5])

The architecture of GLUnix system is shown in Figure 1.4. There are three components inside the system, which are: per-cluster master, per-node daemon and per-application library. Each cluster has a master to coordinate jobs on distributed node, and each job can be chopped into several smaller processes, then sent and executed parallel in different nodes, For example, Prog B started at Node Macbeth, and executed at Node Othello.

The GLUnix Shell is the interface of the global operating system layer, users launch commands from this shell and all these requests will be distributed and processed in the cluster, while still keep the Single Interface Image (SII), users will still have the feeling that these commands are executed locally and integrally. The most obvious similarity between GLUnix Shell and Galaxy shell is that both Shells will execute command remotely, i.e., on a collect of resources. However, the two shells have many different points:

- Goal

  GLUnix was originally intended to support interactive sequential and parallel programs through transparent remote execution and load balancing. Performance is the most concerned issues for GLUnix. While Galaxy is a public resource utility, it involves more complex issues such as addressing, security, service control, and scalability.

- Resource management

  In order to achieve transparent remote execution, GLUnix uses centralized structure to manage resources, represented by a single master node to control a collection of client nodes, this structure is relatively straightforward to design, build and debug. On the contrary, Galaxy doesn't have a centralized node for all resources in the resource pool because its Peer-to-Peer character, all resources can join and leave the resource pool at their discretion, and all related "handshaking" procedures are distributed and autonomous.

- Resource distribution

  GLUnix is targeting a relatively locally distributed and homogeneous machines, normally it is running on a collection of machines inside a domain, such as U.C. Berkeley. GLUnix is extremely suitable for system administrators to test systems in a cluster and for simulation work which need many machines involved. While Galaxy is a public resource utility, it involves outsourcing, addressing and allocating resources; generally these resources are heterogeneous and distributed remotely

1.3.3 CODINE

CODINE is a resource management system created at Florida State University aimed to optimize the utilization of heterogeneous software and hardware in a network environment. Its easy-to-use

GUI eases the life of users and administrators. The architecture of CODINE is shown in Figure 1.5. There are four types of daemons in CODINE that are master, scheduler, execution and communication daemons. CODINE uses a central control module to coordinate the whole system, which are composed by master daemon, scheduler daemon and database, shown in the left hand side in Figure 1.5. The master daemon is the heart of the central control module; periodically it receives information about each node inside of CODINE cluster including workload, job progress, and available resource by the execution daemon running on them, then the master stores this information into the database. The Scheduler is responsible for matching job with available resource, and then it sends the matching list to master, which further sends jobs to specific nodes according to the matching list. At each node, there is one execution daemon, which is not only responsible for executing the job assigned by the master, but also reporting its status at regular interval. The communication daemon at each node is to communicate with the master daemon, either synchronously or asynchronously to make the communication more efficient, fast and reliable.

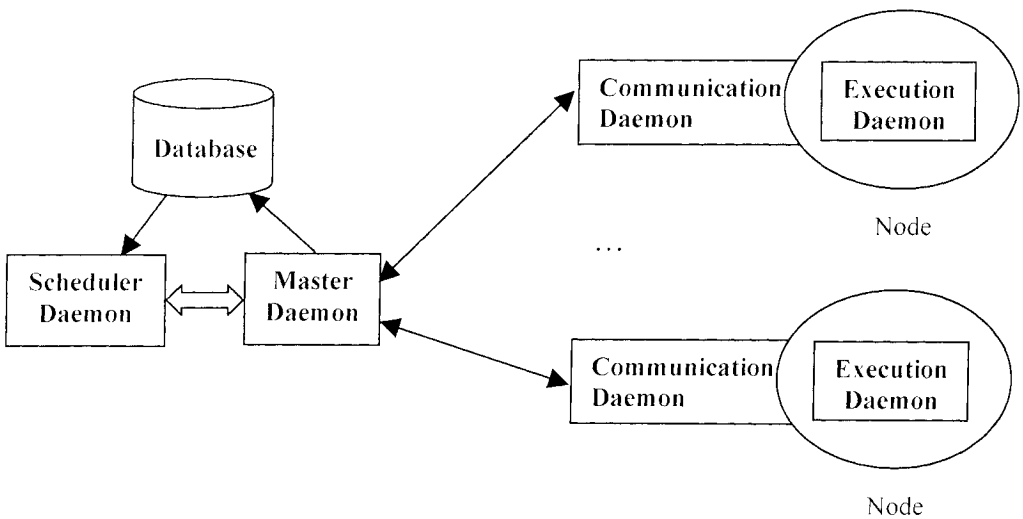

Figure 1.5: CODINE architecture

Because the master daemon is critical, CODINE provides a shadow master functionally. Once the CODINE master fail, a new master will be selected and put into the front line. CODINE provides a Single System Image (SSI) for the whole cluster, the disks attached to cluster nodes appear as a single large storage system and every node in CODINE has the same view of the data. Compare to CODINE, Galaxy is more public which involves more heterogeneous machines all over certain area, thus a strict authentication protocol and security checking are needed. Galaxy also differs

from CODINE by its special type ring algorithm used in it Resource Addressable Network (RAN) which is expecting to improve the efficiency of resource management.

1.3.4 LSF

Platform LSF is the flagship product of LSF Platform Computing based in Canada and it is a leading commercial solution for production-quality workload management. LSF enables good resource usage across corporate LANs. Based on the production-proven, open, grid-enabling, Virtual Execution Machine (VEM)™ architecture, Platform LSF manages and accelerates batch workload processing for compute-and data-intensive applications. LSF adopts high performing and open scalable architecture and its web-based, SOAP/XML interface facilitates the customization and integration of applications. As an industrial product, LSF uses many technology including fairshare, preemption, advance reservation, and resource reservation to create an intelligent scheduling policies. These policies ensure the right resources are automatically allocated to the right users for maximum efficiency. LSF is a self-healing and self-adaptive system which can reduces administration and management requirements and associated costs. LSF also has strong security protection as a mature industrial product. LSF has been implemented on various UNIX and Windows/NT platforms. LSF provides almost the same functionality as CODINE such as load balancing and fault tolerance, and it also provides similar SSI service like CODINE, with an industrial standard. Table 1.1 compares above four related projects with Galaxy by their available services, features, and fault tolerance.

Table 1.1: Five resource management projects comparison

Support Features	Galaxy	GLUnix	CODINE	LSF
Single entry point	No	Yes	No	No
Single file hierarchy	No	Yes	Yes	Yes
Batch support	Yes	Yes	Yes	Yes
Interactive support	Yes	Yes	Yes	Yes
Parallel support	Yes	Yes	Yes	Yes
Load balancing	Yes	Yes	Yes	Yes
Fault tolerance	Yes	Yes	Yes	Yes

In Table 1.1, the single entry point means a user can connect to the cluster as a virtual host. The system transparently distributes the user's connection requests to different physical hosts to balance the load. Galaxy shell doesn't support this because every user in Galaxy has to be a Galaxy node already, and each user has a unique domain address when it joined Galaxy. The single file hierarchy means on entering into the system, the user sees a single, huge files system image as a single hierarchy of files and directories under the same root directory that transparently integrates local and global disks and other file devices. Galaxy doesn't support this feature but use domain addressing to organize the file structure inside Galaxy, thus every user will only be able to see local resources in default, but a user is able to explore or utilize resources on a remote machine in Galaxy if he passed the authentication checking. Like normal Unix shells, Galaxy shell supports both batch and interactive commands to make the life of shell users easier. The batch command can be put in a file and loaded into Galaxy shell when required. Parallel support is a very important function provided by Galaxy shell, through it, a user could broadcast information to multiple nodes inside Galaxy; this is very useful to update the profile of a node or Galaxy type rings. Since load distribution is very critical for the performance of a cluster system, load balancing is also an important function of Galaxy. Load balance is not only an issue at the beginning moment but also at the duration of a job. When a job request was received, Galaxy profiles each node and assigns the most appropriate node to execute the job; and during the execution, Galaxy also watches the load status on the machine, should the load is too high, then Galaxy tries to balance the load to other nodes. The RAN and GRMS provide strong fault tolerance for Galaxy, there is no obvious weak points in the chain which could jeopardize the whole system, and should any node fails, there is always one node takes its place, so the whole Galaxy system is a self-healing system.

CHAPTER 2

GALAXY SHELL DESIGN

Galaxy shell talks downward to the middleware level and provides an upwards interface for the application level. Thus the middleware greatly affects the design of Galaxy shell. The following section will provide a general picture of the middleware layer, and then the design of Galaxy shell will be detailed.

2.1 Galaxy middleware layer

Galaxy is a PCU system, which opens the membership to public resources, by doing this many benefits are achieved such as lowering the cost of participation, preventing monopoly and creating a geographically distributed resource base that is capable of satisfying location specific resource requirements. However, a PCU needs to solve some problems before it can be declared a success, these problems including scalability, trust, security, and incentive control, in addition, PCU also needs to address the frequent arrival and departure of independent resources. Galaxy middleware layer is created for handling these problems. As shown in Figure 1.2, from the Galaxy architecture, we can see that Galaxy middleware layer has three substrates, RAN, GRMS, and Service layer.

Galaxy uses RAN to provide naming and directory service. RAN naming can be divided to two kinds of naming: profile (or type) based naming and positional naming. Profile based naming is based on profiling incoming resource and categorize it to a predefined type. The rational behind this is that there are almost infinite software and hardware combinations for a resource, and if we don't categorized these into a few types, the message overhead created by discovery and dissemination might be too big, we just cannot afford to such a descriptive based naming. Predefined types cut down the size of the RAN messages, eliminate the necessity for complex resource matching algorithms, and decouple the naming from discovery to increase the scalability, thus efficiency is improved, and users can also define their type as the need be. Figure 2.1 shows the mapping from description based naming to profile based naming.

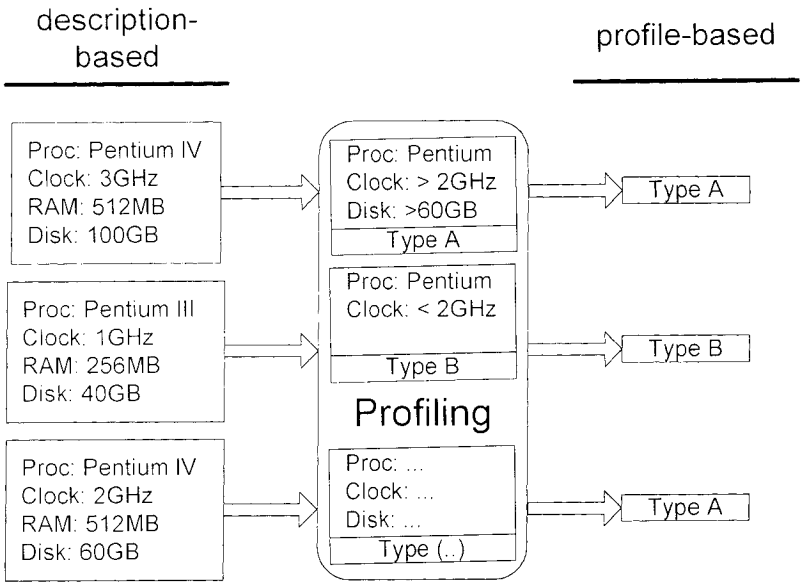

Figure 2.1: Profile based naming

(Taken from "Design of a Quality of Service Aware Public Computing Utility", Maheswaran et al., School of Computer Science, McGill University)

When a resource joins the RAN, it is analyzed to obtain a description of the resource characteristics as a set of attribute-value tuples. This set is then profiled into a resource type. Resources of the same type are collected together in type rings. In the RAN domain, multiple type rings each comprises the resources of a unique type/profile. There are arbitrary links between the rings to avoid cliques of resources being isolated. Positonal names are given to each resource in a ring. A resource in a ring has at least two pointers (route entries) to form the ring (to left and right resources in the ring); Also it can have multiple nonfixed number of pointers to other resources that help route to a destination with a O(log n) number of routing hops and self-healing process.

There are arbitrary links between the rings to avoid cliques of resources being isolated. Within the rings the resources are placed according to their positional names. A resource in a ring has several pointers point to other resources to form a fabric for better communication inside a ring. The RAN rings provide an efficient proximity-aware discovery mechanism.

Positional naming is the other naming method in RAN, this naming is actually based on dynamic information of the current node in the Galaxy system, such as proximity, traffic, and etc. The positional naming enables the RAN to handle QoS as the integral part of the discovery

mechanism. The rational behind the positional naming is that in a dynamic P2P network, it makes no sense to neglect the current network profile, while try to provide an efficient and fair resource distribution. RAN initially adopts the concept of positional naming, and this affects both when the incoming resource joins Galaxy and when a client sends a request using positional naming. Figure 2.2 shows the procedures of a resource joining Galaxy resource pool.

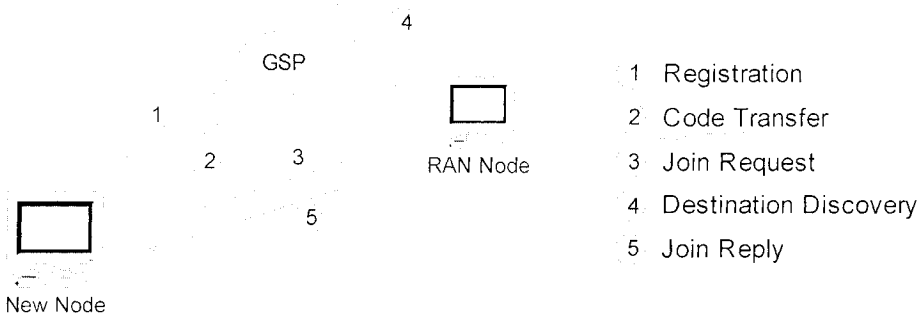

Figure 2.2: Resource joining Galaxy

(Taken from "RAN Naming and Discovery", Balasubramaneyam Maniymaran, 2004 [2])

1. When a resource wants to join the Galaxy, it first contacts the Galaxy service provider (GSP).
2. The latter sends profiling and galaxy bootstrapping modules (Galaxy daemon) to the resource, the profiling code probes the resource for a specific set of resource attributes and categorized the resources into one of the RAN-recognized types based on profiling result. Once profiled, a resource is identified by the profile name (and its positional name).
3. The resource sends the join request to any node inside the Galaxy resource pool.
4. The "entry" node checks the profile of the incoming node, and calculates the place where the resource should be, and put the new resource into the Galaxy.

The searching and binding are also using the same two kinds of naming service, by using type based naming and positional based naming, the overhead of searching a certain resource inside a huge resource pool will be greatly reduced. Galaxy shell, which will be detailed in following sections is heavily involved with these two kinds of naming services.

Another main module in the Galaxy middleware layer is Galaxy Resource Management System (GRMS). It includes many sub modules including resource addressing module, authentication

module, and central control module. GRMS uses a "community-oriented" architecture to manage the resources, which is completely different from traditional resource management architectures such as centralized, hierarchical, or distributed, where a resource is associated with a manager in a static manner. In this community-based architecture, a resource can contact any well standing member of the manager community and receive the management service. The community of resource managers is organized in a P2P overlay and can be located and accessed efficiently by all resource peers.

Galaxy service layer is the highest layer of the Galaxy middleware; these services provide generic capabilities for resource peers to perform activities such as launching resource acquisition commands, managing and monitoring acquired resources, and releasing resources. Currently several services are designed including Galaxy shell, Galaxy Network File System, and Application level QoS management. Galaxy shell is a command-line interface to Galaxy, users can user it to interact with Galaxy system or resources that are allocated by the Galaxy to the resource peers. The design of Galaxy shell will be detailed in the following sections.

2.2 Galaxy shell Rationale

As Galaxy research goes further and deeper, there is a need for a research tool which is able to support Galaxy research. Galaxy shell is designed and implemented for this purpose. Galaxy shell is different from a traditional shell in Unix or Linux workstation. A normal shell we often used in Unix or Linux environment is a command interpreter for the operating system of a standalone workstation. Most commands in a Unix shell are localized commands, which provide an interface for users to communicate with the core of the operating system. Most functionality provided by a Unix shell such as search, pipeline, and redirection are based on the same machine. On the contrary, Galaxy shell is designed for supporting a Public Computing Utility, which provide computer services to allow firms to focus less on administering and supporting their information technology and more on running their business. However, delivering computing service is not that simple like delivering electricity over copper wires, there must exist a mechanism to create context before a success computing utility delivery can be fulfilled. Electricity became a commodity when consumers decided it was no longer in their interest to be power generation experts, and a safe, cheap, reliable and measurable delivery way was there. A Public Computing Utility will success only when those companies that embrace it early will indeed acquire a strategic advantage over those that continue to insist they can do it better themselves. Thus in addition to the normal functionalities often found in a Unix shell, Galaxy shell needs to support

many functionalities specially required by a PCU to create this context. These functionalities include:

- Resources control

  Search, query, bind (reservation), and authenticate resources in the Galaxy resource pool

- Resource consumption

  Execute a task at specified resources (consume the resources), this consummation could be pipelined or multicast, which can help in scalability and processes migration research.

- GRMS optimization

  Investigate the viability of RAN design, routing algorithm, and authentication options.

In order to support these special functions, Galaxy shell was naturally designed to contain three modules, which are shell, local daemon, and peer kernel. The reason to have the three modules came from the requirements of a Galaxy node, it needs communicating with other nodes to search certain node or nodes, join or leave certain type rings, collect and update network profile, all these communication needs are handled by peel kernel. Every Galaxy node provides certain computing services to registered clients, these services can be added or removed as time goes by, there is a need for creating a dedicated module to contain these service, in Galaxy shell, this module is local daemon. Like all other shells in Unix we are already familiar with, Galaxy shell needs provide an interface for users to communicate with the internal services, thus a shell sub component is also created. Following gives a detailed description of the three modules.

The shell module is the interface of the Galaxy shell like normal shells in Unix machines, users use this shell to input command and get respond, the shell also has its parser for analyzing the input commands. The local daemon is the core service provider of a Galaxy node, all service invoked by external commands will be implemented in this module, as the research of Galaxy continues, new services will be added into this module. The peel kernel module is responsible for supporting the unique needs of Galaxy, such as query the network information, search a certain node or a type of nodes based on either static or dynamic constraints, bind or unbind to certain node or certain kind of nodes, and get or set the default domain, type, attributes, and node for reducing user's input work if similar operations are repeating over and over again. Galaxy shell implemented many common functions provided by a Unix shell, the remote procedure call and

pipeline among remote nodes are also implemented, in addition to that, Galaxy shell also reserves many keywords and switch symbols for the special needs in Galaxy research in the future, such as query based on dynamic constraints. However, due to the current progress of GRMS, these functions haven't been implemented yet, but their commands were reserved and it is easy to be hooked up once real implementation of those functions are ready.

There are two options for creating such a developing tool, either a GUI application or a command-line shell. Both have pros and cons. Although command-line shell is difficult to use and learn, it is very flexible and powerful once users have conquered it. Thus a command-line shell project was created. Another import reason for this decision is that implementing a command-line shell is easier to develop than a GUI application.

There were some considerations about where the shell should physically locate. There are three options: outside Galaxy, at GRMS, or at every Galaxy resources. The last one was chosen because first we need to optimize public resources management, such as proximity between resources inside Galaxy resource pool, another reason is that in a P2P system, a resource contributor can and should be an eligible consumer.

2.3 Galaxy shell commands

The issues discussed in Section 2.2 motivated the design of Galaxy shell command. From an analysis of those issues, it can be noted that generally there are two types of tasks, one is involving resource management, including searching, binding and querying resources according certain criteria, or configuring the Galaxy system, we name these commands as "built-in commands" because they are an important part of Galaxy shell which are used to investigate GRMS design and these commands are embedded in the peer kernel module of Galaxy shell; There is another kind of commands which are implemented by executable files on target resources, they can be executed either locally or remotely, also they can be pipelined and/or multicast. We name these type of commands as "external commands" since they are external to Galaxy shell, which can be any executable files and these commands is extendable independent of the Galaxy shell.

Built-in commands don't consume resources but create the condition for external commands to do that. The profile based naming (static type based naming) and position based naming (dynamic

type based naming) is often plugged into built-in commands for advanced searching and reservation. External commands could execute some executable files on a remote node, or execute some methods supported by the local daemon of a remote node. Here comes a security issue: what are the privileges of external commands? Galaxy GRMS has a module which is responsible for authentication, by using it, a flexible scheme could be realized: different access privileges are given to clients according to their identification. GRMS is responsible for creating this protection mechanism.

When considering the syntax of Galaxy shell, the switch symbol for built-in commands was decided to be double dash. The reason behind this is many single dashes have been widely used in Unix shell, and since Galaxy shell will provide many commands similar to those commonly seen in Unix, it would be better to avoid such conflict from the very beginning, and double dashes is also easy to parse.

Galaxy shell commands involved much design concepts in Galaxy middleware, and both built-in commands and external commands are extendable as Galaxy research continues. The commands of Galaxy shell are detailed in Chapter 3 "Galaxy Specification".

2.4 Galaxy shell architecture

To implement the mechanisms discussed in Section 2.2, Galaxy shell needs to talk to RAN for searching and binding, and needs to send request to either locally or remotely. When a resource joins Galaxy, it downloads and installs Galaxy daemon from Galaxy Service Provider. Galaxy daemon includes three parts: shell (pure I/O interface), local daemon (for inbound request), and peer kernel (for searching, binding, querying, and etc). A modular design is shown in Figure 2.4. This modular design assigns a clear-cut responsibility to each component, thus it is easier to develop, debug, and maintain.

From Figure 2.4, when a user enter a command from the shell, it will first be sent to the parser inside the shell, if there is some syntax error, an exception will be thrown to ask the user to revise input. If no error detected, the command will be processed. If the redirection parameter of the command is not the local machine and it is a built-in command, then the command will be sent to the peer kernel to search or bind remote resources. If the redirection parameter of the command is not the local machine and it is an external command, then the shell will launch a remote method

invocation by a way supported by Galaxy system, which is XML-RPC. Before this remote method invocation could fire, the shell needs to get the network information of Galaxy resources (where the resource is, what the profile it is, and etc), and make reservations, all these preparation works will be done by using built-in commands. If the redirection parameter of the command is the local machine or missing, then the local daemon at the local machine will be responsible for processing it. Table 2.1 lists all scenarios of this process.

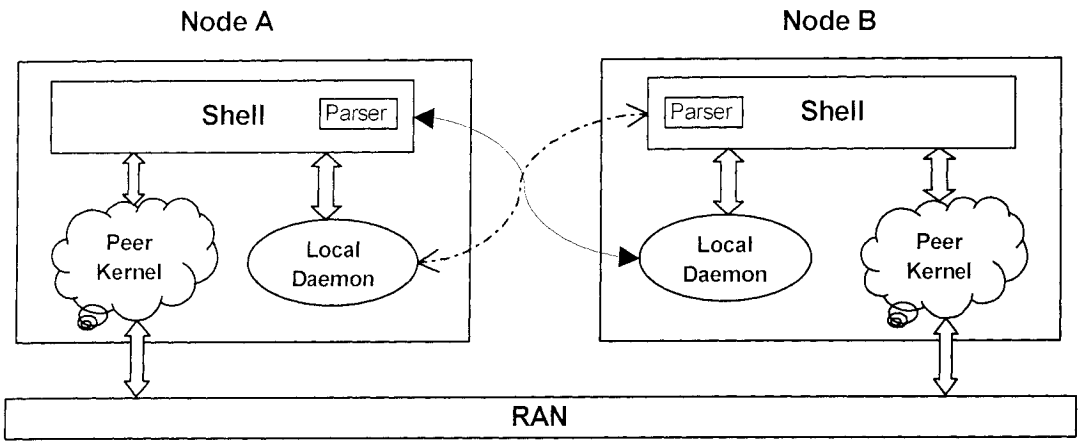

Figure 2.3: Galaxy shell Architecture

Table 2.1: Galaxy shell process scenarios

Command	Redirection parameter	Process module
built-in command	local	peer kernel of local machine
	remote	peer kernel of remote machine
external command	local	local daemon of local machine
	remote	local daemon of remote machine

2.4.1 Shell

Here the shell is a narrow definition that means the pure input/output framework inside Galaxy shell (see Figure 2.4). The shell operates in a simple loop: it accepts a command, interprets the command, dispatches the command to corresponding modules to execute, and then waits for

another command. The shell displays a ">" prompt, to notify users that it is ready to accept a new command. Most shell commands are not built into the shell, but are dynamically loaded and started by the shell framework when they are invoked. Separating the shell framework from the commands enables developers to dynamically add new commands to the shell. The parser is a sub module inside the shell. When the user types a command at the prompt, the parser reads the command line and breaks the line into tokens, for lexical analysis. If everything is fine, corresponding modules will be called to fulfill the command.

2.4.1.1 Parser

There are many parser generators available for implementing the parser, the most common ones are Lex, Flex, Yacc and Bison. Lex is the lexical analyzer supplied for many years with most versions of Unix. Flex is a freely distributable relative associated with the GNU project. Yacc is a parser generator developed at Bell labs. Bison is a freely distributable implementation associated with the GNU project. JavaCC, an open source project, which stands for Java Compiler Compiler is used to generate the token manager and parser for Galaxy shell. Table 2.2 compares these parser generators.

Table 2.2: Comparison of parser generators [29][30][31]

Parser generator	Language of parser generated	Work style	Grammar accepted	Specialties
JavaCC	Java	top-down	EBNF	• Common token actions • Special token rules • More rules
Lex/Flex	C	top-down bottom-up	EBNF	look ahead in the input stream past the end of the matched token
Yacc/Bison	C	bottom-up	BNF	LALR grammar

JavaCC and Lex/Flex are actually quite similar. Both work essentially the same way, turning a set of regular expressions into a big finite state automaton and use the same rules. The big difference is the Lex and Flex produce C, whereas JavaCC produces Java. In addition, JavaCC have some

nice features that Lex and Flex lack, such as common token actions, MORE rules, and SPECIAL TOKEN rules which simplify the parser generation.

There is a bigger difference between Yacc and JavaCC in that Yacc works bottom-up while JavaCC only works top-down [31]. This means that Yacc and Bison make choices after consuming all the tokens associated with the choice, whereas JavaCC has to make its choices prior to consuming any of the tokens associated with the choice. However, JavaCC's lookahead capabilities allow it to peek well ahead in the token stream without consuming any tokens; the lookahead capabilities reduce most of the disadvantages of the top-down approach. In addition, Yacc and Bison reads BNF grammars while JavaCC accepts EBNF grammars. In a BNF grammar, each nonterminal is described as choice of zero or more sequences of zero or more terminals and nonterminals. EBNF extends BNF with looping, optional parts, and allows choices anywhere, not just at the top level. For this reason Yacc/Bison grammars tend to have more nonterminals than JavaCC grammars and to be harder to write. Generally speaking, it is often easier to write semantic actions for JavaCC grammars than for Yacc grammars, because there is less need to communicate values from one rule to another. Yacc has no equivalent of JavaCC's parameterized nonterminals.

Since Galaxy is implemented in Java, JavaCC is suitable for this project. JavaCC is not intuitive to grasp but proved to be a powerful tool once it has been conquered. The workflow of JavaCC is like this: the programmer supplies a collection of "Extended BNF production rules"; JavaCC uses these productions to generate the parser as a Java class. These production rules can be annotated with snippets of Java code, which is how the programmer tells the parser what to produce.

2.4.2 Local daemon

The role of local daemon is someone like a "receptor" and "waiter" in a hotel. It checks the reservation of a request, authenticate the request, if passed, then it provides the service for the request. By creating local daemon module, the work for developing, debugging and maintaining are greatly eased. Local daemon also has an important function to mask the difference of heterogeneous resources. Since there are so many different hardware and platforms out there, it will be very difficult to implement a shell to talk to these systems without a module to provide a

kind of abstraction at the remote side. Local daemon provides a unique interface and an abstraction for shell.

Local daemon has a sandbox to protect illegal access, and provides VM control for a resource. When a request arrives, local daemon will first check its identification, and reservation ticket. If everything is fine, then it will create a thread or process to take care of the request according to the agreement specified in the reservation ticket. There is a main thread always running to supervise the request process behavior, it has the power to suspend or even kill a child process should some abnormal things happen. The main thread later can resume the child process should the situation back to normal.

2.4.3 Peer kernel

Peer kernel provides the updated Galaxy information. Its main responsibilities includes periodically send out heartbeat to Galaxy GRMS to report the node's profile. The interval of this network updating work should be carefully considered: too frequently is expensive and unnecessary, on the contrary, if too sparse, the information of this resource on the RAN could be stale. Another major function provided by peer kernel is providing an API for GRMS, which could be used by the shell. Normally this API includes searching, binding, querying, and reserving resources inside Galaxy resource pool. The shell doesn't need to understand how the GRMS works as long as it knows the API. This modular design separates the task between front-end and back-end programming, and provides the convenience for Galaxy development.

2.5 Galaxy shell workflow

Some issues need to be addressed when designing the Galaxy shell, such as shell session control and communication between remote nodes. Following sections will talk about some of these issues.

2.5.1 Bootstrap

When the shell starts, some information will be loaded into the shell, including some configuration information and network information. There are several locations to put this information: memory, disk, and both according to the properties of this information. It seems that the third way is attractive because some information such as shell configuration or preference set by a user should not be deleted once a session finished, so put them into the disk and load them into the shell at next session will make the shell more friendly and convenient. On the other hand, some information are dynamically changed, because Galaxy system is not static, so in order to let the shell use most up-to-date information, some information must be stored in the memory for fast accessing and updating. This dynamic information will not be preserved once user terminates a session (close the shell).

2.5.2 Remote procedure call

Since Galaxy is a distributed system, the communication method of distributed machines is an important factor which could greatly affect Galaxy design and performance. Currently, there are several common distributed communication technologies are using: (Java) RMI, Corba, XML-RPC, and Servlet. Table 2.3 compares these technologies.

RMI is a Java-centric distributed object system. It inherits all of the benefits of Java. An RMI system is immediately cross-platform; any subsystem of the distributed system can be relocated to any host that has a Java virtual machine handy. However, the only way currently to integrate code written in other languages into a RMI system is to use the Java native-code interface to link a remote object implementation in Java to C or C++ code. This is a possibility, but painful. The native-code interface in Java is complicated, and can quickly lead to fragile or difficult-to-maintain code. The speed is also a bottleneck for RMI since an additional interpretation layer is added to the processing of instructions. The Java just-in-time compilers (JIT) are capable of generating native instructions from Java bytecode, but there is still an additional piece of overhead in running each piece of Java code.

CORBA is a popular protocol for writing distributed, object-oriented applications. It's typically used in multi-tier enterprise applications for integrating legacy systems. CORBA is well supported by many vendors and several free software projects. CORBA is designed to be language-independent. Object interfaces are specified in a language that is independent of the

actual implementation language. This interface description can then be compiled into whatever implementation language suits the job and the environment. CORBA is a more mature standard than RMI, including comprehensive high-level interfaces for naming, security, and transaction services. Unfortunately, CORBA is very complex. It has a steep learning curve, requires significant effort to implement, and requires fairly sophisticated clients. It's better suited to enterprise and desktop applications than it is to distributed web applications. Another concern about CORBA is that the adoption of CORBA is shrinking in industry [32], CORBA itself may become a legacy technology.

Table 2.3: Comparison of three distributed computing technologies

Distributed computing technology	**Language dependent**	**Platform dependent**	**Complexity**	**Speed**	**Overhead**
RMI	yes	no	low	Slow	low
CORBA	no	yes	high	Fast	low
XML-RPC	no	no	low	fast	high

XML-RPC is a lightweight way to make procedure calls over the Internet. It converts the procedure call into XML document, sends it to a remote server using HTTP, and gets back the response as XML. CORBA forces you to explicitly define interfaces for types, while XML-RPC doesn't have this requirement. XML-RPC just uses URLs for to reference objects, this shows its flexibility or simplicity. XML-RPC is language and platform independent, which make it quite suitable for Galaxy system, with so many heterogeneous systems collected. However, the XML data format adds overhead compared to CORBA's binary format. In Galaxy, it is highly probable that a resource will provide a legacy services implemented in a language other than Java, and there is the possibility that certain sub modules of local daemons will need to be implemented in different languages, thus XML-RPC is the best candidate for distributed computing in Galaxy.

2.5.3 Process control

There are some considerations about dealing with the dynamically changing Galaxy resource pool. What would happen if the situation changed before or during a remote procedure call which means the remote process couldn't be completed? A cheap way is just neglect and give up; but this way contradicts with the Galaxy system goal, which is to provide an accessible, stable, and reliable public resource utility. Galaxy uses this way to address this problem: before a client uses a remote resource, it must make a reservation first, the reservation will specify when, where, who, and how the resource will be used. Once a reservation has been made, the resource will try its best to honor this agreement. By this "reserve-first" strategy, we can avoid chaos from the beginning. However, this method is not flawless, first it is complex and expensive, it needs the complex GRMS system to support this function; second, even with reservation, there is no absolute guarantee the resource will honor that promise because the resource could be dead due to a power blackout. Although this scenario should be rare, GRMS needs to take care of it, maybe a backup resource could be called to replace the failed resource, and GRMS should migrate the job and restart it on the backup resource. All these issues need to be further investigated in the GRMS design.

Remote computing often affected by the network performance, it is highly probable that a remote connection is jammed or even broken during a transaction. In order to solve "hanging" problem, every remote method should be fired by a child thread instead of the main thread of the shell, and a timeout will be raised should a remote method call doesn't return for a long time, and user can always be assured that the control is returned to the shell.

CHAPTER 3

GALAXY SHELL SPECIFICATION

Galaxy shell extends the functions of normal shells by adding special commands supporting resource searching, binding and consuming. Resource management system (RMS) is the backbone behind the interface, and the resource naming system used by RMS is reflected in the syntax of Galaxy shell.

3.1 Galaxy resource naming system

Galaxy resource naming system is the core component of RMS. RMS is the fundamental component which manages the resources and provides low-level functionalities such as naming and discovery to the upper level components like Galaxy shell. All the machines are profiled when they join Galaxy, and their types are known at that time, which can be used as the principle means for searching. At present time, there are three categories of naming, which are group based, static type based, and dynamic type based naming.

3.1.1 Group based naming

Galaxy resource pool is divided by many domains, each domain represents a collection of machines who have some common properties. The common properties can be but not limit to location, political or interests approximation. Group based naming is a basic naming according to the domains. This is kind of like an extension of peer groups in JXTA, where each group represents certain trust domain, where in each domain, some common services are provided to its members.

3.1.1.1 Usage

The syntax of domain based naming is:

[globalName]:[domainName]:[machineName]

Where globalName is the resource pool name, domainName represents the name of certain domain, machineName is the destination machine where the command will be directed to and executed there, and all three fields are optional. Two wildcards can apply to any field: "?" matches any and "*" matches all. If a field is missing, then it means this field will be instantiated by all available items in this field, equal to the wildcard "*".

Example 1:

Galaxy:CanadianInstitute:Computer_A

This redirection parameter will direct a Galaxy shell command to "Computer_A", which is located the CanadianInstitute domain of Galaxy resource pool.

All contents inside square brackets are optional, for example, if no domain name presents, then the command applies to all domains inside galaxy, if no MachineName presents, then the command applies to all machines inside the domain.

Example 2:

Galaxy:CanadianInstitute:*

This redirection parameter will direct a Galaxy shell command to all machines inside CanadianInstitute domain.

All running result should be returned to Galaxy shell console except for pipeline command.

Example 3:

Galaxy:CanadianInstitute:

This redirection parameter has the same effect as the previous one.

Example 4:

Galaxy::

This redirection parameter will direct a Galaxy shell command to all machines inside Galaxy, no matter which domain those machines are located in. It should be noted here that "*::*" is syntax correct but meaningless since at current time, Galaxy resource pool is unique.

3.1.2 Static type based naming

Static type based naming is based on the profiling result when a resource join Galaxy. The profiling result is a list of attribute-value pairs. Here static type means the attributes are based on static properties, such as CPU make, disk volume, physical memory, operating system, and etc. Due to so many hardware combinations for computers in the world, it is better to sort those combinations into several predefined categories or types for easy management. There is a fine line between too general and too fine for these categorize actions. A comprise has to be made for easy management and high type recognition ratio. For those resources who don't fall into any predefined type, they will be tagged "unrecognized type" in the resource directory in their domain.

3.1.2.1 Definitions

3.1.2.1.1 Static type

A static type is a user defined resource collection; each type is a list of attribute-value pairs. For example, one type could be like following:

```
Type name: WorkStation
----------------------------------
CPU = Pentium
OS = Windows
Memory = 512M - 1024M
Disk = < 100 G
Netspeed =  < 1 Gbps
...
```

Static types are stored in a special directory named "Type directory", which provides a predefined list of types for a certain domain.

Static types can be defined by using command "deftype", the syntax is:

deftype --t <typeName> <attribute-value pairs>

For example, a type "Work Station" can be defined using this command:

deftype –t WorkStation CPU=Pentium&&OS=Windows&&Disk<100G&&Netspeed=1Gbps

3.1.2.1.2 Type directory

Type directory is a typing naming service which stores predefined or user defined types, each domain has its independent type directory for scalability reason, a shell can send a command to the type directory service to query current available types inside the domain. Normally, there are a few predefined types stored in the directory, while this directory can be expended as the need be.

3.1.2.2 Usage

The syntax to use type based naming is:

<built-in command> [<group based naming>] [<search constraint>]

Where,

<built-in command> are commands specially involved with Galaxy resource searching and binding, they are:

lstype

List all types which satisfy given search constraint (if any), inside the group(s) specified in the given group based naming (if any).

bind

Bind all nodes which satisfy given search constraint (if any) and given duration constraint (if any), inside the group(s) specified in the given group based naming (if any). Here bind means to build a connection channel between the shell and the local daemon of the nodes to be bound.

<group based naming> has the same syntax and meaning introduced in previous section, this item will give the scope where the built-in commands work on. If it is absent, then it means the scope is in the default domain, the default domain is initially set to the domain where the shell is sitting in.

<search constraint> can be either static type search constraint or dynamic search constraint, the latter one will be introduced in following section. If the search constraint is a static one, then it is either an optional filter based on certain attribute-value pairs or certain predefined types, plus an optional number constraint and reservation constraint, which specify how many nodes need to be searched or processed and how many seconds (duration) the bind will last, default is one. The syntax of <search constraint> is:

<search constraint> ::= <static search constraint> |
<dynamic search constraint> |
<number constraint> |
<reservation constraint>
<number constraint> ::= --n <number of nodes will be searched>
<reservation constraint> ::= --d <number of seconds the resource will be reserved for>
<static search constraint> ::= --t <predefined type constraint> |
--a <attribute constraint> |
<predefined type constraint> ::= <type name>
<attribute constraint> ::= <attribute constraint> |
<attribute constraint> <logic operator> <attribute constraint>
<logic operator> ::= '&&' | '||'
<attribute constraint> ::= <attribute> <relation operator> <value>
<relation operator> ::= '<' | '<=' | '>' | '>=' | '='

It should be noted that here "--t" and "--a" are using double dash, the reason is to avoid conflict with normal shell command switches. Following are several examples to show how the type based naming is used.

Example 1:

lstype Galaxy:CanadianInstitute:*

Will list all types available in Domain CanadianInstitute, like:

```
Type: Server_1
-----------------------------------------------------------------
CPU      | Memory | Disk  | OS      | NetworkSpeed |  ...
Pentium | 1024M  | 100G  | Windows | 100 Mbps     |  ...

Type: WorkStation
-----------------------------------------------------------------
CPU      | Memory | Disk  | OS      | NetworkSpeed |  ...
Pentium | 512M   | 10G   | Linux   | 10 Mbps      |  ...
...
```

Example 2:

lstype Galaxy:: --a cpu=pentium

Will list all types available in every domain inside Galaxy resource pool, the result will something like this:

```
Domain: Domain _1     Type: Server_1
-----------------------------------------------------------------
CPU      | Memory | Disk | OS      | NetworkSpeed |  ...
Pentium | 1024M  | 100G | Windows | 100 Mbps     |  ...

Domain_1   Type: WorkStation
```

```
-------------------------------------------------------------
CPU      | Memory | Disk | OS       | NetworkSpeed |  ...
Pentium | 512M   | 10G  | Linux    | 10 Mbps      |  ...

Domain_2   Type: Server_2
-------------------------------------------------------------
CPU      | Memory | Disk | OS       | NetworkSpeed |  ...
Pentium | 1024M  | 100G | Windows | 100 Mbps     |  ...

Domain_2   Type: Palm
-------------------------------------------------------------
CPU      | Memory | Disk | OS       | NetworkSpeed |  ...
Pentium | 100M   | 1G   | Windows | 100 Mbps     |  ...
...
```

Example 3:

bind Galaxy:CanadianInstitute: --a cpu=pentium&& mem<1024M --n 3

Will bind three nodes in Domain CanadianInstitute inside Galaxy resource pool, where their CPU are Pentium and memory are less than 1024M. The result could be something like this:

```
3 machines bound!     Domain: CanadianInstitute
-------------------------------------------------------------
Machine 1, Alias=Ying
Machine 2, Alias=Mahes
Machine 3, Alias=Mayni
```

Example 4:

bind Galaxy:CanadianInstitute:? --a cpu=pentium&& mem<1024M

Will bind one machine in Domain CanadianInstitute inside Galaxy resource pool, where its CPU is Pentium and memory is less than 1024M. The result could be something like this:

```
1 machine bound!      Domain: CanadianInstitute
-------------------------------------------------------------
Machine 1, Alias=Ying, Type: WorkStation
```

Example 5:

bind Galaxy:CanadianInstitute:* --t palm --n 10

Will bind ten machines whose types are palm in the domain CanadianInstitute, the result could be something like:

```
10 machines bound!    Domain: Galaxy:CanadianInstitute:*
-------------------------------------------------------------
Machine 1: alias: Ying
Machine 2: alias: Mahes
Machine 3: alias: Maniy
...
```

Example 6:

lsbind

Will show current bind information, the result could be something like:

```
Bound information
-------------------------------------------------------------
Machine Alias    Domain           Type
Ying             Galaxy:mcgill:   laptop
Mahes            Galaxy:ubc:      desktop
```

Example 7:

unbind ying

Will unbind Machine Ying, the result could be something like:

```
Machine ying is unbound!
```

All above examples just show some light on how to use static type based naming in the Galaxy shell commands; detailed information about the Galaxy command will be elaborated in following section.

3.1.3 Dynamic type based naming

Dynamic type is related to dynamical information which is collected by Galaxy Resource Management System (GRMS) from time to time. This information could be network topology, load capacity, throughput, QoS, and location approximation. The syntax of extended type based naming is using certain options to represent certain combinations of dynamic profile. These options could be but not limited to network speed, proximity measuring, load, and etc. Notes of these options are summarized in Table 3.1:

Table 3.1: Dynamic based naming options

Notes	Meaning	Optional Value
--l	Load	heavy | medium | low
--dt	Desired Throughput	heavy | medium | low
--p	Proximity measuring	far | medium | near
--q	QoS-constraint	d (dedicated resource) | b (best effort)
--s	Speed	fast | medium | slow

Example 1:

bind Galaxy:CanadianInstitute: --p nearest --l light

Will bind one machine which is closest to the machine hosting the shell and with light load. If there are multiple machines satisfied with the condition, only the first one found will be returned to the shell.

Example 2:

bind Galaxy:CanadianInstitute: --q d

Will bind one machine which will be dedicated to current shell.

3.2 Galaxy shell Commands

Galaxy shell commands can be divided into two types, which are built-in and external commands, combination of these commands provide the basic functionality of Galaxy shell. These commands provide an interface for users to access the service provided by Galaxy shell and resource daemon. All Galaxy commands are case insensitive like Unix Shell.

3.2.1 Built-in commands

Built-in commands are the core of Galaxy shell which provide the basic functionality such as set and get environment, searching, binding and other "house keeping" operations. Built in commands themselves don't consume resources but do the preparation work for external commands to do so.

The complete syntax of built-in commands is:

<Built-in command> [domain based naming] [<search constraint>]

Where,

<search constraint> is optional and used to set certain condition to execute a command, the explanation of <search constraint> is the same as previous text.
<Built-in command> is a set of special commands which support plumbing work such as configuring, searching, binding/unbinding in Galaxy shell, they are:

- lstype

This command will list the type information inside given resource pool and/or domains

- deftype

This command will define a new type

- bind

This command will bind machine(s) who satisfied given constraint, these constraints could be either a static type constraint or dynamic type constraint

- unbind

This command will unbind the machine with the given machine name

- lsbind

This command will list bind information of current shell

- setdomain

This command will set the default domain for all following commands in current session, thus after the domain was set, if the domain based naming doesn't give the domain name, then the default domain name will be used. For example

setdomain domain_1
lstype

The second command will list all types inside domain_1 of galaxy resource pool

- getdomain

This command will return the default domain name for current setting. The initial default domain is the domain where the node hosting the shell is in.

Example 1:

setdomain domain_1
getdomain

The second command will return "*domain_1*"

Example 2:

*setdomain **
getdomain

The second command will return "*", means all domains available are going to apply for following commands.

- setnode

This command will set the default node for all following commands, thus if the default node was set, then all commands following without specific node name will use the default node. The default node of default node name is "*", which means all nodes available. For example

setnode node_1
lstype galaxy:domain_1:

The second command will list the type of node_1 inside domain_1 of galaxy resource pool

- settype

This command will set the default type for the constraint of all following commands.

- gettype

This command will return the default type was set

- setattr

This command will set the default attribute for the constraint of all following commands. For example,

setattr cpu=duron&&mem>500M
*lstype galaxy:domain_1:**

The second command will return all types of nodes, which are inside domain_1, and satisfied with the default constraint set by the first command.

- getattr

This command will return the default attributes for the constraint of all following commands.

3.2.2 External commands

External commands are those commands does a specific task, such as listing files in certain directory, executing an executable file, and etc. External commands can be executed either locally or remotely, depends on the node specification following the command. If the node specification is missing or it equals to localhost, then this command will be assumed as a local command, which will direct to local daemon of current node, otherwise, it will be treated as a remote command, which will invoke a remote method sitting in a remote machine, in this case, the shell will send the request to the local daemon of the remote machine. Before a remote external

command being fired, the remote machine has to be bound to current shell by using bind command. This bind operation will create the link between the local daemon of remote machine.

The syntax of external command is:

<external command> |@<node name>| |<argument list>|

Where,

<external command> is normal Unix shell command, and the names of executable files.
<node name> is the alias of certain machine, which was acquired from searching command. The default node is the current machine which is hosting the shell. The default node will be applied if the node name is not specified.
<argument list> is the argument list for external commands.

For example,

ls @mimi -l

This command will list all files and directory at Machine mimi with long format.

3.2.2.1 External commands list

Following aware the list of the external commands currently supported by the Shell and local daemon of a Galaxy resource.

cat writes the contents of the files to the standard output

clear clears the screen

cd changes the directory with respect to the current directory

cp makes a copy of the file

date prints the date

echo writes the parameters of this command to the standard output

exit exits the shell

grep searches for a pattern in the files and prints the lines which matched the pattern

help gives a help on a command

ls gives a listing of all files in the directory

mem gives the total memory available to the java runtime system and free memory

mkdir creates a subdirectory with the given name

mv moves the source file to the destination file

pwd prints the current working directory

rm deletes given files from the current directory

time gives the current system time

wc gives the count of number of lines, words and characters in a file

version prints the current version of the shell

pipe pipe (|) is not really a command. It is a way of joining two commands together. It is similar to Unix pipe.

The external command list can be added as the need be, the extension of supporting external commands will be detailed in Chapter 5.

3.2.3 Batch command

Since the main goal of Galaxy is to support RAN research, it is possible a user has to input many commands with much repetition, batch commands will ease this stretch by automate some routing commands serial, just like a script file of UNIX Shell. When a user type the name of a batch file, the script file will be loaded and its content will be executed line by line like user input them manually in the same order.

3.2.4 Remote pipeline

Galaxy shell supports remote pipeline operation, here remote pipeline is different from traditional pipeline in Unix shell where the pipeline is executed locally. Galaxy shell supports a "global-like" pipeline, which means that the result of execution of one command at Machine A can be directed to another command at Machine B. The pipeline can be divided to two types, sequential and concurrent pipelines, based on single casting or multi casting requirement of Galaxy shell.

3.2.4.1 Sequential pipeline

Sequential pipeline is similar to tradition pipeline in Unix except the pipeline applied to several distributed machines. The syntax of sequential pipeline is:

<pipeline element> ~ <pipeline element> ~ ...

where,

<pipeline element> ::= <external command> [@<node name>] [<argument list>]

For example,

ls @mimi –l | wc @willy

This command will execute 'ls' on Machine 'mimi' with parameter '-l', then send the execution result to the 'wc' command on Machine 'willy'. The final result (if any) will be returned to the shell who launching this command initially.

3.2.4.2 Concurrent pipeline

Concurrent pipeline has the similar functionality as sequential pipeline but adding multicast function. The syntax of concurrent pipeline is:

<pipeline element> { <multicast list> }

where,

<pipeline element> ::= <external command> [@<node name>] [<argument list>]
<multicast list> ::= <pipeline element> ';' <multicast list> ';' <pipeline element>

For example,

ls @mimi { wc@willy; cat@nova}

This command will execute 'ls' on 'mimi', then multicast the result to machines 'willy' and 'nova', where 'wc' and 'cat' get executed there, correspondently. Then all running result (if any) will be directed to the shell.

3.2.5 Batch command

Since the main goal of Galaxy is to support RAN research, it is possible a user has to input much of commands with huge repetition, batch commands will ease this stretch by automate some routing commands serial, just like a script file of UNIX Shell. When a user type the name of a batch file, the script file will be loaded and its content will be executed line by line like user input them manually in the same order.

3.3 Syntax of Galaxy shell Command

3.3.1 Reserved Words

The following words are reserved and have a special meaning to the Galaxy shell when they are unquoted:

if	then	else	end	in	case
do	for	while	until	function	~
|	||	:	*	;	?
{	}	@			

3.3.2 BNF for Galaxy shell

The following is the syntax of Galaxy shell in Backus-Naur Form (BNF), which the parser of Galaxy shell is based on.

<letter> ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z|
A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

<digit> ::= 0|1|2|3|4|5|6|7|8|9

<number> ::= <digit> |
<number> <digit>

<word> ::= <letter> |
<word> <letter> |
<word> '_'

<word_list> ::= <word> |
<word_list> <word>

<resource_pool> ::= <word> | '*' | '?'

<domain_name> ::= <word> | '*' | '?'

<node_name> ::= <word> | '*' | '?'

<redirection> ::= <resource_pool> ':' [<domain_name>]':'[<node_name>]

<option_name> ::= '-'<word>

<option_value> ::= <word> |
<option_value> ' ' <word>

<option_element> ::= <option_name> |
<option_name> ' ' <option_value>

<option_list> ::= <option_element> |
<option_list> ' ' <option_element>

<type_name> ::= <word>

<type_constraint> ::= '--t '<type_name>

<attribute_name> ::= <word>

<relation_operator> ::= '<' | '>' | '<=' | '>=' | '=' | '<>'

<attribute_value_pair> ::= <attribute_name> <relation_operator> <attribute_name>

<logic_operator> ::= '&&' | '||'

<attribute_value_pairs> ::= <attribute_value_pair> |
<attribute_value_pairs> <logic_operator> <attribute_value_pair>

<attribute_constraint> ::= '--a'<attribute_value_pairs>

<static search constraint> ::= <type constraint> |
<attribute constraint>

<number constraint> ::= --n <number>

<reservation constraint> ::= --d <number>

<qos value> ::= 'b' | 'd'

<qos constraint> ::= '--q' <qos value>

<speed value> ::= 'fast' | 'medium' | 'slow'

<speed constraint> ::= '--s' <speed value>

<proximity value> ::= 'far' | 'medium' | 'close'

<proximity constraint> ::= '--p' <proximity value>

<load value> ::= 'heavy | 'medium' | 'low'

<desired throughput constraint> ::= '--dt' <load value>

<load constraint> ::= '--l' <load value>

<dynamic search constraint>::= <load constraint> |
<qos constraint> |
<speed constraint> |
<proximity constraint> |
<desired throughput constraint>

<search_constraint> ::= <static search constraint> |
<dynamic search constraint> |
<number constraint> |
<reservation constraint>

<built_in_command_keyword> ::= 'lstype' | 'bind | 'unbind' | 'lsbind' | 'settype' | 'setnode' |
'setdomain' | 'setattr' | 'gettype' | 'getnode' | 'getdomain' | 'getattr' | 'deftype'

<built_in_command> ::= <built_in_command_keyword> |
<built_in_command_keyword>' '<redirection> |
<built_in_command_keyword>' '<redirection>' '<search_constraint>

<external_command_keyword> ::= <word>

<node_alias> ::= <word>

<relocation> ::= '@'<node_alias>

<external_command> ::= <external_command_keyword> |
<external_command_keyword>' '<relocation> |
<external_command_keyword>' '<relocation>' '<option_list>

<command> ::= <built_in_command> |
<external_command>

<command_list> ::= <command> |
<command_list> ';' <command>

<multicast_list> ::= '{'<command_list>'}'

<galaxy_command> ::= <command> | <galaxy_command> '|' <command> | <command>
<multicast_list>

CHAPTER 4

GALAXY SHELL IMPLEMENTATION

The implementation of Galaxy shell is divided into the implementation of the three sub modules of Galaxy shell, which are the shell (pure interface), local daemon, and peer kernel. In the following sections, the implementation of these three sub modules of Galaxy is introduced, at the end of this chapter, the implementation of XML-RPC server, which is used to test the remote procedure call from Galaxy shell, is also introduced.

4.1 Shell

Shell is responsible for inputting and digesting rudimentary commands, pipes, I/O redirection, and background processing. It works with local daemon and peer kernel to fulfill the mission in Galaxy. Inside shell, there is a parser module, which is responsible for tokenizing and parsing incoming commands.

The Shell simply loops through getting a command from the user, evaluating that command, and returning an output. *gshell* interacts with the user and simply go to the appropriate class when it needs to perform certain actions. *gshell* is like a bus that picks up people that need to go to certain places. *gshell* delivers them there, waits for them to do their thing at the stop and then gets back a changed person. The program design matches this cycle, the *gshell* class handles the inputting from the user. It then asks a command class in either local daemon or peer kernel to run the command, and the output is passed back to *gshell*, outputted, the cycle starts again. Figure 4.1 shows the file structure of the shell module.

Figure 4.1 shows the file structure of shell module of Galaxy shell. From the figure, we can see that there is a sub package parser inside gshell package, demonstrates the hierarchical relation between them. There are 13 classes directly inside gshell package, among them; there are three interfaces, which are *ConsoleLine.java, OutputWatcher.java, and ProcessWatcher.java*. Following is a short description of each of these interfaces.

- *ConsoleLine.java:* an interface used between the console, and the shell, this interface specifies what will be called when the user presses the return key inside the console.
- *OutputWatcher.java:* an interface that allows a class to be notified every time something is written to an output stream.
- *ProcessWatcher.java:* a simple callback interface that allows classes which use the *threadedcommand* class to know when their spawn processes have terminated.

Figure 4.1: File Structure of shell module

Beside three interfaces, *gshell* package has ten class files which working together to implement the shell functionalities. Several key class files are shortly introduced as followings.

- *gshell.java*: The core file of this package, it creates the main frame and the text area for the console. It parses the commands, then dispatch them to corresponding modules to process.
- *Console.java*: It extends the text area, and provides a way to read input and write output. It will also handle the keyboard output.
- *EventOutputStream.java*: It implements an output stream in which the data is written into a byte array. The buffer automatically grows as data is written to it. In addition to this, whenever something is written to the byte array that information is also sent to the object that created it. The classes implementing the shell commands will write to *System.out*. This class provides a way to capture that stream and redirects to the text area.

- *Help.java*: This class provides a way to view the help. It extends *JFrame* class, and it contains a text area and a button for each command. By clicking the button the corresponding file is read and displayed on the text area.

Among all these files inside *gshell* package, the Class *Console* is playing a key role in displaying input and output during the interact process. The UML diagram of this class is shown in Figure 4.2, we can see that *Console* class implements *KeyListener* class, extends *JTextArea* class, and references *dimension, String, StringBuffer, and Vector* classes. *Console* class also associates with *ConsoleLine* interface and is referenced by *gshell* class.

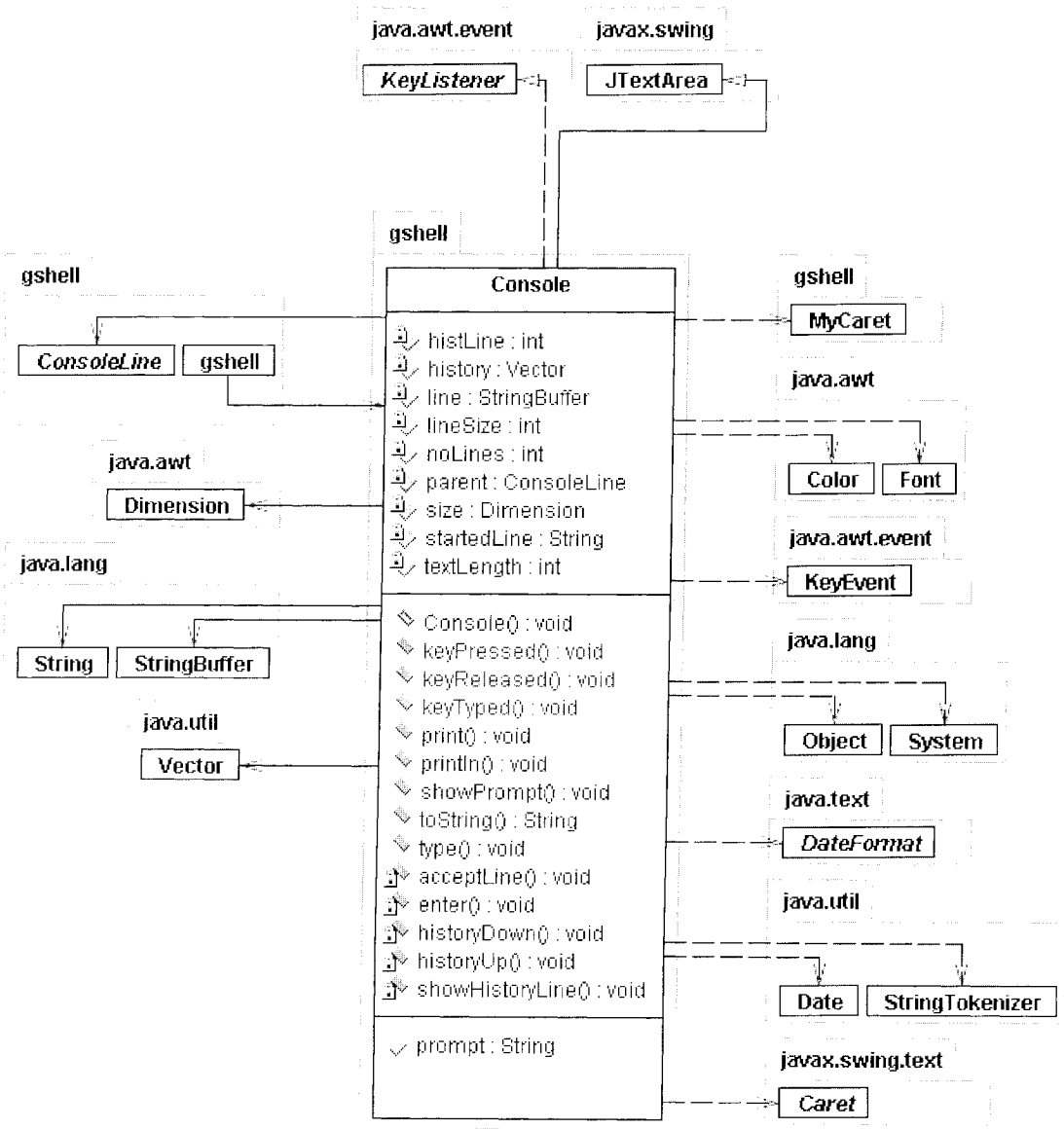

Figure 4.2: UML diagram of Class *Console*

4.1.1 Parser module

Parser module is a sub module inside Shell module, it is responsible for doing lexical analysis and parse out the commands. JavaCC, stands for "Java Compiler Compiler", is an open source project is used to generate the parser module. JavaCC reads a description of a language and generate code, written in Java that will read and analyze that language. Although the Galaxy commands are not complex as a language, it is not bad idea to use an efficient and elegant parser instead of messy string comparison, and by adopting a parse will make the testing, debugging, and maintaining work easier in the future. Although we can write a parser manually, it is difficult if the input contents have a complex structure. Fortunately, JavaCC is able to help by generating a parser automatically unless the programmer provides a seed file (.jj file). This technology originated to make programming language implementation easier; hence the term "compiler compiler" comes out.

In order to produce the parser, which normally includes a token manager in it, a .jj file needs to be provided by programmers. The .jj file specifies a collection of Extended BNF rules which is used to break the sequence of characters into a sequence of tokens. It is often a headache sometime when there is more than one regular expression matches a prefix of the remaining input. JavaCC adopts following rules for picking which regular expression to use to identify the next token:

- The regular expression must describe a prefix of the remaining input stream.
- If more than one regular expression describes a prefix, then the regular expression that describes the longest prefix of the input stream is used, so called "maximal munch rule".
- If more than one regular expression describes the longest possible prefix, then the regular expression that comes first in the .jj file is used.

After those tokens are produced, the parser consumes the sequence of tokens, analyses its structure, and do further process. What kind of process the parser will do is up to programmers who specified this in the .jj file also. From this description, we can see that JavaCC is completely flexible.

JavaCC is a program generator. It reads a .jj file and if that .jj file is error free, produces a number of Java source files. Figure 4.3 shows the files generated by JavaCC, they are:

- *SimpleCharStream.java* — represent the stream of input characters
- *Token.java* — represents a single input token
- *TokenMgrError.java* — an error thrown from the token manager

- *ParseException.java* — an exception when the input failed to be parsed
- *Parser.java* — the parser class
- *ParserTokenManager.java* — the token manager class
- *ParserConstants.java* — an interface associating token classes with symbolic names

Figure 4.3: File Structure of parser module

The *Token* class represents a token produced by the token manager. Figure 4.4 shows the UML diagram of this class. It can be noted that each token object has some fields to store its properties, such as the location of the token in the seed file, the type (*kind*) of the token and what the next token is. The *Token* class associates itself due to the linked list structure, and is referenced by *ParserException* and *Parser*. The *Token* class is also being depended by *ParserTokenManager* class, where a change to the *Token* class will affect *ParesrTokenManager* class.

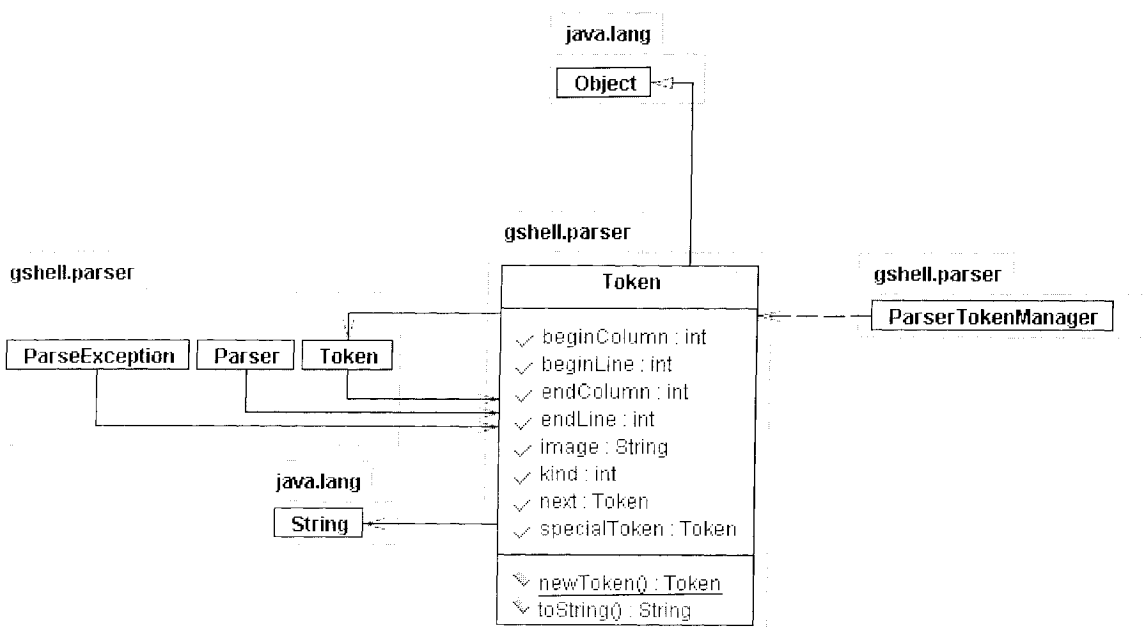

Figure 4.4: UML diagram of Class *Token*

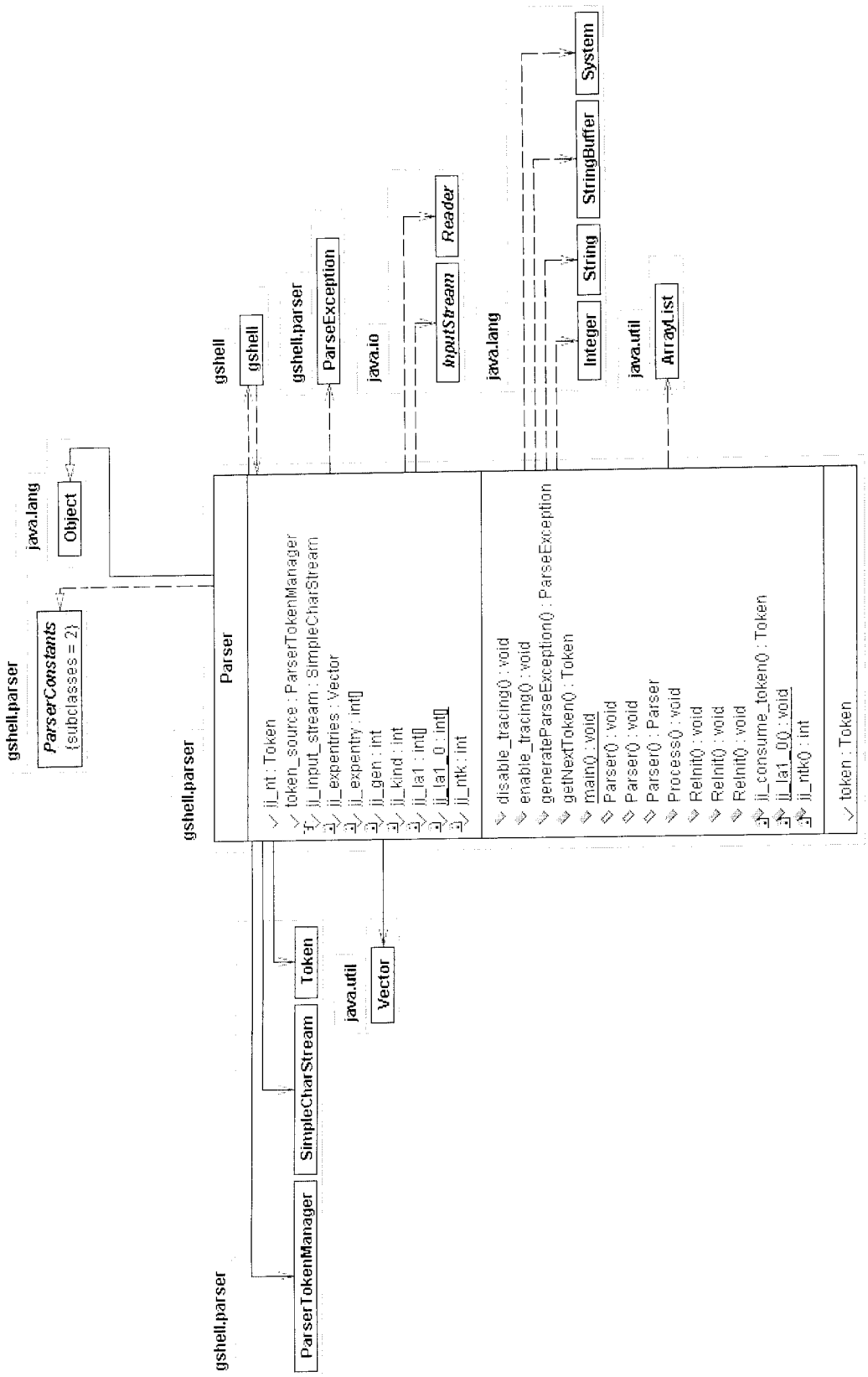

Figure 4.5:UML diagram of Class *parser*

The *parser* class is in the central position in the *parser* module; it associates *ParserTokenManager* and *Token* class for consuming tokens. Figure 4.5 shows the UML diagram of the *parser* class. There is a public method *Process* is responsible for consuming the incoming tokens. It recognizes these tokens, make certain change if necessary, and then load certain command class file to execute the command. We can also see that there is a circle dependencies between *parser* and *gshell* class, any changing in one party will automatically affect the other one.

4.2 Local daemon

Local daemon module is the place where the service is provided. When a resource joins Galaxy, the local daemon is always running on that resource, and waiting for incoming commands. Thus each resource in Galaxy is acting like a service provider, the special thing is that each resource in Galaxy is acting both a service provider and a consumer. Local daemon can accept commands coming from both locally or remotely. For remote commands, local daemon will authenticate its identity and checking its reservation ticket before providing the service, local daemon also provides a sandbox like protection mechanism against malicious clients. At the time when this shell was implanting, the research of those security functionalities is still on going, thus these functionalities are not implemented in this version. Figure 4.6 shows the file structures of *localdaemon* package, and Figure 4.7 shows the UML diagram of class *LocalDaemon*, which will run as a server to provide services.

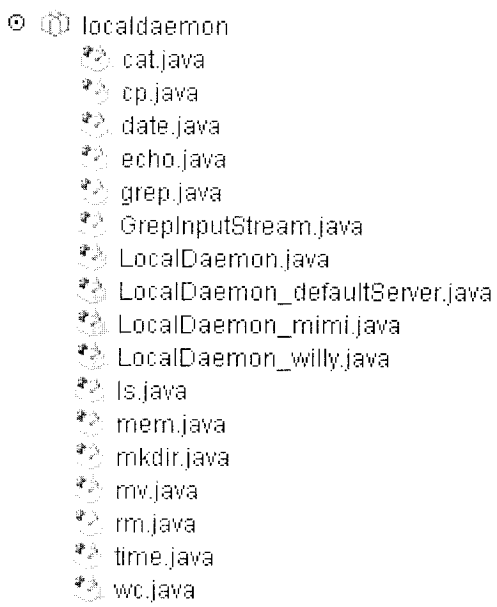

Figure 4.6: File Structure of local daemon module

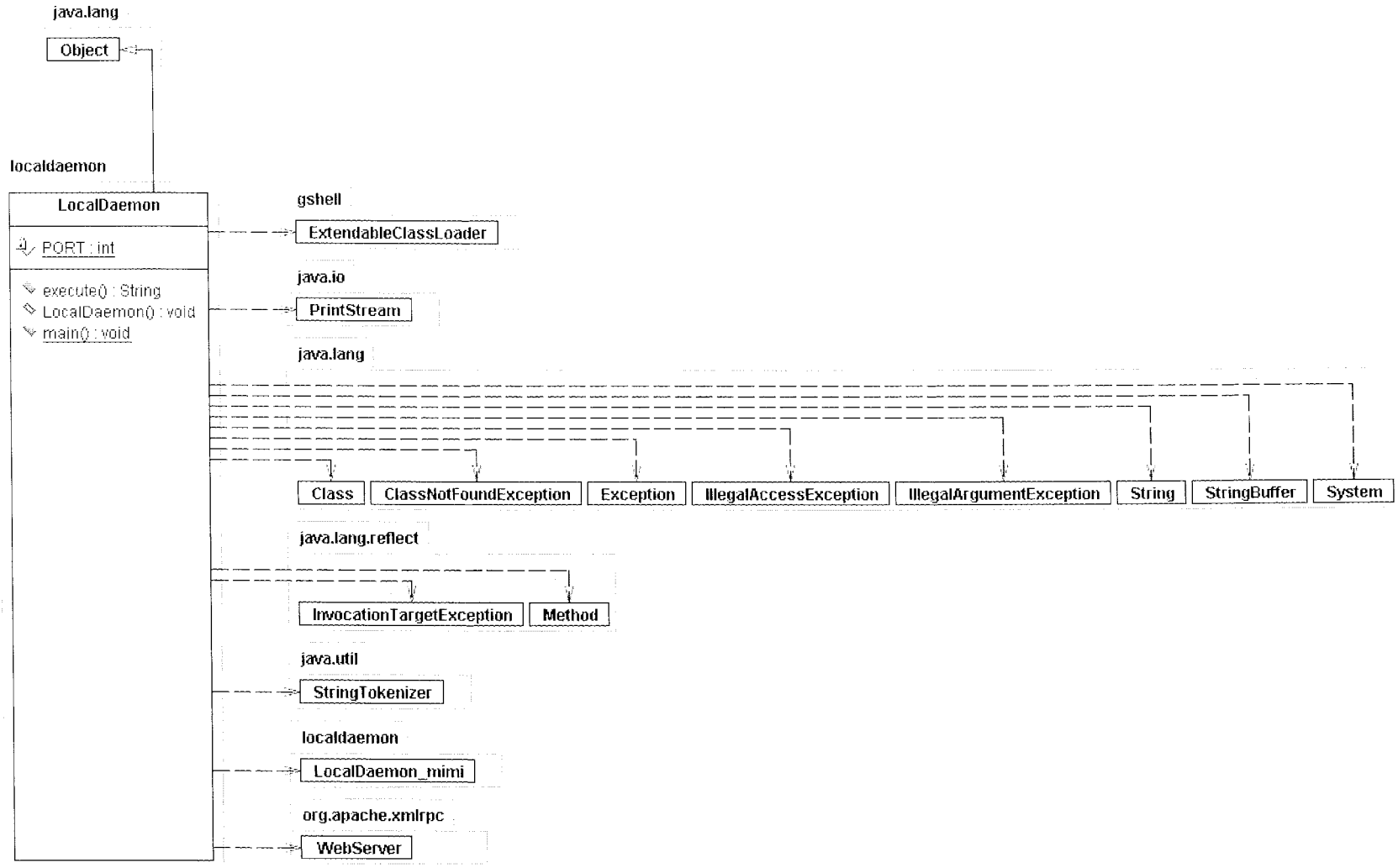

Figure 4.7: UML diagram of Class *LocalDaemon*

From Figure 4.7, we can see that the class *LocalDaemon* has an instance variable *PORT*, by assigning different numbers to this field, a simulation of distributed resources can be created at one machine, which is convenient for development. *LocalDaemon* associates with *ExtendableClassLoader*, while the latter one is able to load and execute a command class file on the fly. If a command class cannot be found during loading, a *ClassNotFoundException* will be thrown. *WebServer* class in the *org.apache.xmlrpc* package is referenced for starting a simple XML-RPC server. The *execute* method is for executing inputting commands.

4.2.1 Local daemon commands

Currently, twelve command classes have been implemented in the *localdaemon* package. Each of these classes represents a command which can be loaded and executed on the fly. All the following classes will contain the main method. Each class executes a command specified by name itself. The main method will take an array of strings as arguments that are given to the respective command. All the commands will write the output the *System.out*, which will be directed to the console of the Galaxy shell. Following is the short description of these command classes.

- *cat.java* — prints the commands contents of the given file
- *cp.java* — copies the source file to destination file
- *date.java* — prints the current system date with the following format month, day, year
- *echo.java* — writes each given string argument to the standard output
- *grep.java*— searches the given string in the files specified
- *ls.java*— lists the filenames and folders in the current directory
- *mem.java*— displays the total memory for the java console and also the available free memory
- *mkdir*.java— creates a directory, if it does not already exist
- *mv.java*— move or rename a file from current directory
- *rm.java*— delete a file from current directory
- *time.java*— prints the system time
- *wc.java*— prints the lines, words, characters in the given file or input stream

The implementation of above methods is straightforward, just need to write a class with expecting input parameters, and do the corresponding task equivalent to the name of the class, and return a

string, which will be shown on the shell console. Figure 4.8 shows the UML diagram of the command class *ls*.

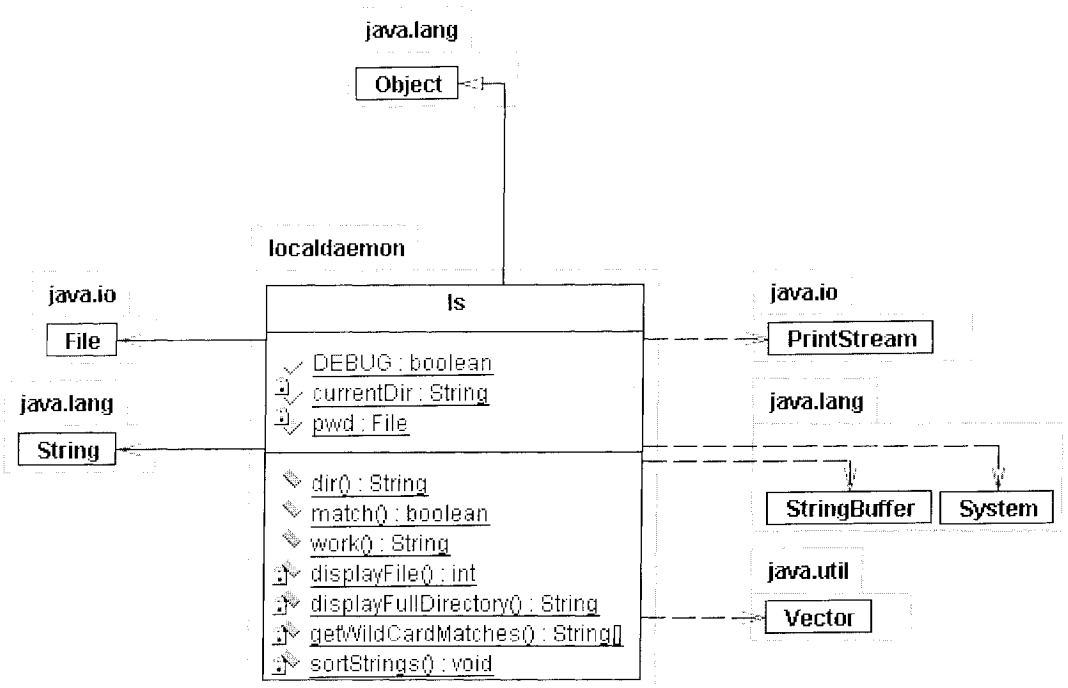

Figure 4.8:UML diagram of Class *ls*

It should be noted that local daemon should not only be able to load and execute a command class written in Java, it also needs to be able load and run an executable file written in other languages. For this situation, Java Native Interface (JNI) can be used to make the cross language operations. At current version of the Galaxy shell, this is not implemented.

4.3 Peer kernel

Peer kernel provides an API for the shell module, masking the complexity of GRMS. Through this API, the shell module (client) can explore the Galaxy network, and also advertises itself to other nodes in the Galaxy resource pool. Figure 4.9 shows the file structure inside *peerkernel* package.

Figure 4.9: File Structure of peer kernel module

4.3.1 Peer kernel commands

Currently, there are thirteen command class files are implemented, which can generally categorized into two sub types, configuration commands and networking commands. The implementation and operation of these command files are similar to those commands in local daemon module. A short introduction of these two kinds of commands class files is listed in following text.

Command class files for configuration

- *setattr.java and getattr.java* — set/get the default attribute value pair constraints
- *setnode.java and getnode.java* — set/get the default redirection destination
- *settype.java and gettype.java* — set/get the default type constraints
- *sestdomain.java and getdomain.java* — set/get the default domain constraints

Command class files for networking

- *lstype.java* — list available type information (under given constraints)
- *bind.java and unbind.java* — connect/disconnect a collect of resources
- *lsbind.java* — list current bind information
- *deftype.java*— define a new resource type

4.4 Remote Procedure Call

The remote procedure call between distributed Galaxy nodes is implemented based on XML-RPC, a popular protocol that uses XML over HTTP to implement remote procedure calls. Apache XML-RPC is a Java implementation of XML-RPC, current version is 1.2, which is integrated into the Galaxy shell project to implementing distributed computing. By using Apache XML-RPC, programmer can be abstracted from the detailed XML-RPC protocol implementation while focus on the more important work. Programmers don't need to know how the XML document is parsed and how the HTTP protocol is implemented, the only things need to know is what need to be transferred, and what the interface to call.

Apache XML-RPC supports SAX 1.0 and can therefore be used with any compliant XML parser. The default parser is John Wilson's MinML Parser which is included in the package so programmers don't need anything else to start using the software. MinML is an ideal parser for XML-RPC because it is small, fast, and implements exactly the features of XML which are used by XML-RPC. Following sections will introduce the implementation of Apache XML-RPC in Galaxy, both in server and client side.

4.4.1 XML-RPC Server

On the server side, there are two ways to plug in XML-RPC module: one is to embed the XML-RPC library into an existing server framework, such as Tomcat or WebLogic, or use the built-in special purpose HTTP server. The XML-RPC library comes with its own built-in HTTP server. This is not a general-purpose web server; its only purpose is to handle XML-RPC requests. However, it is good enough for testing the remote procedure call between two nodes. Thus, this built-in HTTP server is adopted. The HTTP server can be embedded in any Java application with a few simple lines:

```
webServer webserver = new WebServer (port);
webserver.addHandler ("examples", someHandler);
```

This built-in HTTP server also provides a useful function, which can set the IP addresses of clients from which to accept or deny requests. It can be used in the authenticate module in the local daemon for advanced resource management. This is done via the following methods:

```
webserver.setParanoid (true); // deny all clients
```

```
webserver.acceptClient ("192.168.0.*"); // allow local access
webserver.denyClient ("192.168.0.3"); // except for this one
```

Before an XML-RPC server to provide a service, it must know how to map incoming requests to actual methods. This is done by registering handler objects to the server like following:

```
addHandler (String name, Object handler);
removeHandler (String name);
```

4.4.2 XML-RPC Client

Apache XML-RPC provides two client classes.

- *org.apache.xmlrpc.XmlRpcClient* uses java.net.URLConnection, the HTTP client that comes with the standard Java API
- *org.apache.xmlrpc.XmlRpcClientLite* provides its own lightweight HTTP client implementation.

XmlRpcClient provides the full HTTP support such as proxies or redirect while *XmlRpcClientLite* is lightweight buy may outperform *XmlRpcClient* in some scenario. Both client classes provide the same interface, which includes methods for synchronous and asynchronous calls.

Using the XML-RPC library on the client side is quite straightforward. Following is an example:

```
XmlRpcClient xmlrpc = new XmlRpcClient ("http://localhost:8080/RPC2");
Vector params = new Vector ();
params.addElement ("some parameter");
String result = (String) xmlrpc.execute ("method.name", params);
```

It should be noted that in the above code snippet, the support to the types of parameters are limited in current library of Apache XML-RPC. In order to pass some unsupported objects, they have to be first wrapped into an object supported, such as a Vector. This is a little bit clumsy but works.

4.5 Galaxy shell extension

Since Galaxy Shell is designed in modular. Galaxy Shell is easier to be extended to match the pace of Galaxy research. Each command supported by Galaxy Shell is implemented by a Java class, thus to add a new command or change an existing command is just related to a single class. There are two steps to add a new command: first is to change the parser to make the parser recognize the new command and process it, secondly, to add a new Java class to implement this command. There are two places to put the new class file, depends the property of the command: if the command is a built-in command, then it should be put into the peer kernel module; if it is an external command, then local daemon module is its destination. Under the help from JavaCC, changing the parser it not very difficult, the only thing need to change is the .jj file, which is located in the gshell.parser package, after changing has been made, running javacc parser.jj to generate the new parser of Galaxy Shell.

In the future, it is very likely that Galaxy shell needs to support remote parallel pipeline. The concept was created but in this version, this concept is not implemented yet. Following gives some suggestion for implementing this function in the future. The syntax of the remote parallel pipeline can be chosen under the developer's will. For example, *ls~wc@willy;wc@nova*. This command means the output of *ls* command again local machine will be used as input of the *wc* commands running on two remote machines, *willy* and *nova*. The semicolon can be used to separate the multiple remote machines. In order to implement this, the semicolon symbol must be added into the parser seed file for successful parsing the input. A vector is recommended to store the multiple commands and destinations, in this case: *wc@willy* and *wc@nova*. The *processCommand()* method in gshell class should be the place need to be changed and also the method *runCommand()*.

CHAPTER 5

GALAXY SHELL USAGE AND ANALYSIS

Galaxy Shell can be deployed on any machine with JVM running on it. Galaxy Shell supports many commands similar to those in common Unix shells but also adds more functionality on it.

5.1 Galaxy shell usage

5.1.1 Galaxy Shell installation

The Galaxy project is in a folder named *gshell*, this folder contains source code, compiled files, supporting libraries, and *help* documents. In the root directory of this folder, a *readme* file contains the basic information about how to run the project. Several batch files are provided for running the project easily on a Windows machine. Executable files are stored in the folder *classes*, in order to run Galaxy Shell; it is a good idea to include this directory into the class path of the computer environment setting. The three sub modules of Galaxy Shell can be found in the *classes* folder, and supporting libraries are put in the *lib* folder. Folder *bak* stores the backup copies of the project. Figure 5.1 shows the file structure of Project Galaxy Shell.

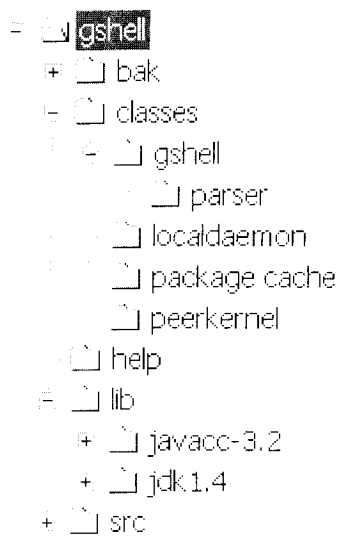

Figure 5.1: File Structure of Galaxy Shell Project

5.1.2 Running Galaxy Shell

When run the Galaxy Shell, open a shell under Unix or a Windows command processor under Windows machine, *cd* to the *classes* folder, and enter ***java gshell***. If the library path is correctly set, then a Galaxy Shell console will be displayed on the screen. After entering into the shell, a prompt *$USER* will remind the user that Galaxy Shell is fully initialized for user with identity equals *$USER*, and is ready to accept commands. Galaxy Shell is designed to be able to use interactively or in a batch through a script file. Should an invalid input is given, the shell will throw a parser error, and tip the user what could be wrong, and wait there for the user to input a new command. If there is no syntax error in command line, but some parameters are missing for successfully executing the command, the user will be notified about the usage of specific command. At any time, user can use the get help information about specific command by typing *help command_name* or use the drop-down menu at the top of the console. When the user wants to close the shell, he/she can just type *exit* to gracefully terminate the session.

5.1.3 Starting XML-RPC server

XML-RPC servers are embedded into the Galaxy Project folder for easier distribution and installation. In order to develop and test the remote procedure call on the same machine, currently the server name of all servers are set to *localhost*, but with different port numbers to simulate different machines. This can be easily changed when real remote procedure calls are to be tested. The Galaxy shell has been tested successfully on *mimi* and *willy* servers of McGill University. It also was tested on LAN in the planet lab of McGill University. The testing analysis will be detailed in next section. Each XML-RPC server is a sub component of a running local daemon of a remote resource. To run a XML-RPC server, just open a shell under Unix or a Windows command processor under Windows machine, *cd* to the *classes* folder, and enter ***java localdaemon.LocalDaemon***, then the XML-RPC server is running and ready to accept requests at a default port which is 8080, this port number can be changed according to users' will. Once servers are running, it is the time to test the remote method call, such as *ls@mimi~wc@willy*. This command will list contents under the project root directory at Server *mimi*, and send the result to Server *willy*, where *wc* (word count) method will be executed, finally, the result will be sent back to the shell console. It is should be noted that the pipeline sign is "~" instead of "|" which latter is often seen in Unix shells, the reason behind this is that in Galaxy shell, the "|" is reserved for local pipeline only, and "~" is used only for pipeline between remote machines.

5.2 Galaxy Shell performance

The resource utilization of Galaxy Shell is small, compared with similar shells. Table 5.1 shows the resource consumption of several shells. These data are collected from a Windows machine with all shells installed, the total physical memory of the testing machine is 512 Mb, and the CPU is Pentium IV with frequency 2.4GHz. All the values in the table are the peak values when they were running at normal status. It can be seen that Galaxy Shell doesn't consume much system resources, as both its CPU and memory consumption are quite low. This attribute makes it suitable for those systems with limited system resources such as PDAs.

Table 5.1: Resource consumption of shells

	CPU (Peak value)	Memory (kb)	Disk (kb) ***
Galaxy Shell	1%	14600	175
JXTA Shell *	3%	29900	306
Windows Command Interpreter **	1%	800	367

* JXTA Shell version: 2.3-pre-16 157e 06-15-2004

** Microsoft Windows XP Professional, Version 5.1.2600 Service Pack 1 Build 2600

*** Not considering support libraries

The speed of Galaxy Shell or the response time of Galaxy Shell is also tested at different circumstances. Figure 5.1 compares the response time of internal commands and external commands of Galaxy Shell inside a LAN. Internal commands are those commands that only have connections with local system and are expected to have shorter response times compared to external commands, where remote procedure calls are invoked. The test was executed on several machines linked in a LAN at McGill University. The link bandwidth between computers is 10Mbps and all testing machines are running Linux systems. Two machines were used for this test, one is a server, and the other is a client. The server machine is running on Linux system, where XML-RPC server is running on it, the client machine is running on Windows XP system, and it sent out requests to the server machine. Ten commands were selected out from both internal and external command sets for testing, and each command was executed 1000 times in a loop, then the average time was record for analysis. From Figure 5.1, the average response time

for internal commands is 8.223 milliseconds, while the corresponding time for external commands is 19.398 milliseconds. The slowdown is mainly due to the serialization and de-serialization during remote procedure calls and network transfer. However, the response time is quite acceptable from a human being's point of view.

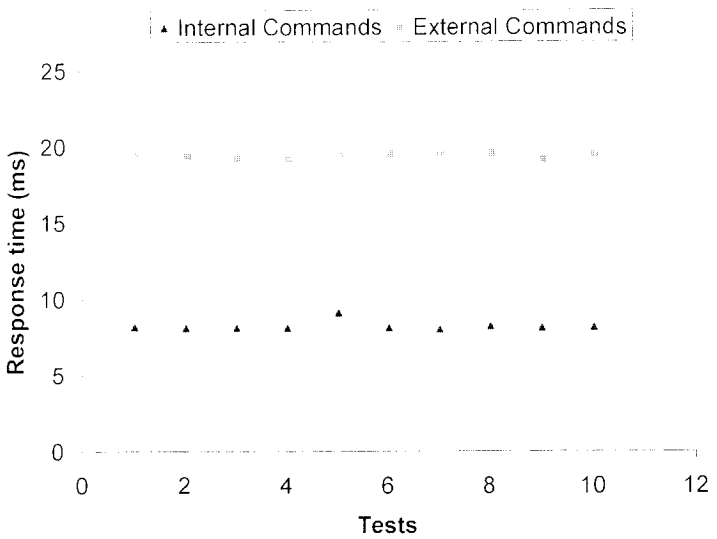

Figure 5.2: Response time of commands in a LAN

(Tested at Planet Lab at McGill University, two machines were used; one is Linux system running as a server, the other one is Windows system running as a client. The two machines are linked in a LAN, where bandwidth is 10 Mbps)

Figure 5.3 compares the response time of internal commands and external commands of Galaxy Shell in PlanetLab network, an overlay network supported by a collection of academic, industrial, and government institutions for research and development. Three nodes in the PlanetLab network were used for testing the response time of external commands, the three nodes are: *node-1.mcgillplanetlab.org*, *node.howard.edu*, and *planetlab1.cs.dartmouth.edu*. All the three nodes were Linux machines and a dedicated slice was allocated on each of the three nodes for doing research without disturb by other users. Among the three nodes, the machine *node-1.mcgillplanetlab.org* was physically located in the same room where the test was starting, while the other two nodes were a little bit far away, they were all in the north-east of America. The connection among PlanetLab nodes is varied just like the machines used. Thus it pretty well represents the miscellaneous and complex network in the real world, which is also the target of Galaxy system. Two machines were used for this test, one is a server, and the other is a client. The server is running on the three PlanetLab nodes, where XML-RPC server is running on it, the

client machine is running on a randomly picked Linux machine in the Planet lab of McGill University, the client machine sent out requests to the server machine. Ten commands were selected out from both internal and external command sets for testing, and each command was executed 1000 times in a loop, then the average time was record for analysis. Since the response time of internal commands is relatively constant, thus only one series of their response time is given in three test groups.

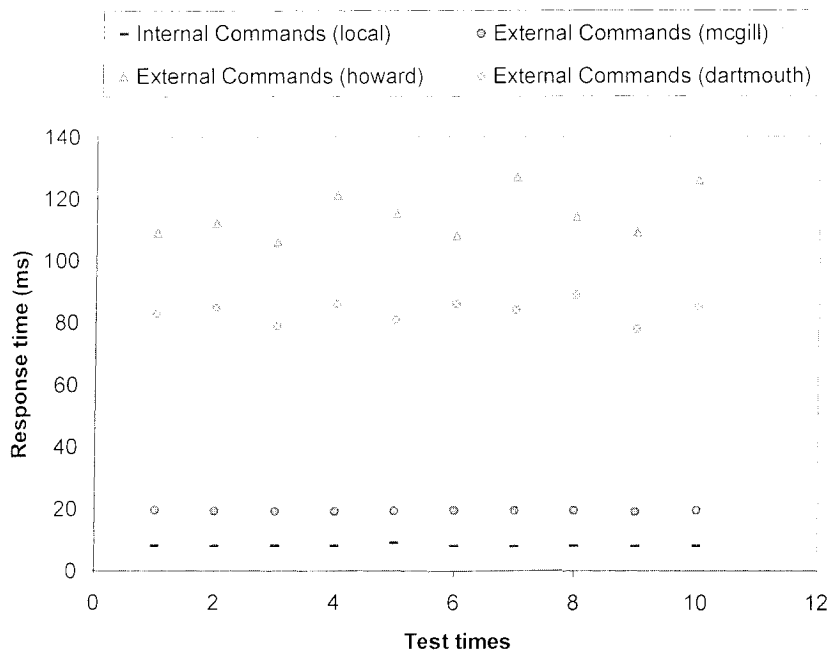

Figure 5.3: Response time of commands in PlanetLab network

(Tested at Planet Lab at McGill University, three Linux nodes in the PlanetLab overlay network were chosen to be servers, the client is a Linux machine in the McGill computer lab which has access to internet)

From Figure 5.3, the average response time for internal commands is 8.223 milliseconds, while the corresponding time for external commands is varied from around 19.4 milliseconds to 115 milliseconds. The response time pretty well demonstrated the distance between the client and server machines. Because the Planet node at McGill is physically close to the client machine, thus a quite quick response is understandable; while the Planet node at Howard University is physically located in Washington DC area, thus it has the longest response time. The Planet node at Dartmouth is in Massachusetts, thus its response time is sitting in the middle. All the response time is quite acceptable and these tests proved that Galaxy Shell is acceptable for distributed computing tasks in Galaxy systems.

As a PCU system, Galaxy needs to support hundreds or even thousands machines simultaneously. At the planet lab, the scalability of Galaxy Shell was tested to explore the feasibility of its design and implementation. We expect the Galaxy Shell has reasonable good scalability. Due to the facility limitation, it was difficult to test the scalability is a real world situation where various network connection and thousands machines working together, so two simplified tests were executed: one is based on a LAN in McGill, the other is based on PlanetLab at McGill. Figure 5.4 shows the general idea about how the broadcasting time was recorded. To communicate with multiple XML-RPC server instances at the same time, the client thread created equal number of child threads and delegated the task to those child threads, then waited for the last child thread returns. The time from the broadcasting method was sent to the last child thread returns was record as the index of scalability evaluation.

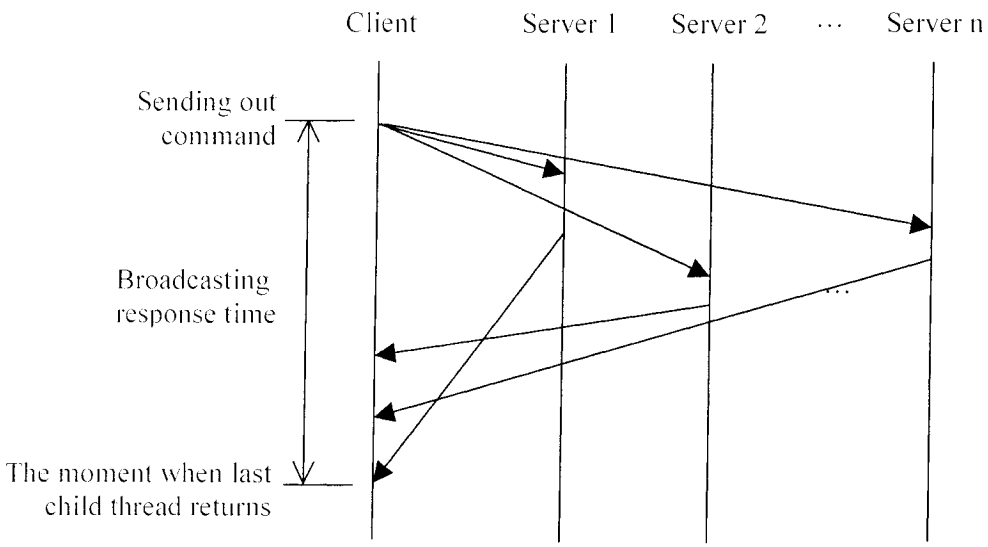

Figure 5.4: Response time of broadcasting

The result of scalability testing in a LAN was shown in Table 5.2 and Figure 5.5. Table 5.2 shows the different response times for executing broadcasting commands to remote machines. Three groups of tests were executed and it shows that the response times were quite small when the number of nodes was below one hundred; when number of nodes reached one hundred, the response time was still less than one second, which is quite acceptable in real situation. When number of nodes was increased to 250, it required about 2.4 seconds to finish the broadcasting execution, and when number of nodes was 500, the response time was little longer at around 5 seconds. It needs more than 10 seconds to broadcast to 1000 machines, this is quite long for

normal users' tolerance, this suggests that current Shell design will perform very well when simultaneously communicating with 100 machines or less and its performance goes down after this threshold. Figure 5.4 shows the response time increase trend as the number of nodes are increased, it can be seen that when the machine number is less than 100, the response time increase is quite slow, and when the number of nodes passes 250, the response time increase much faster than before.

Table 5.2: Response time of broadcasting in a LAN

Machine numbers	Test 1 (ms)	Test 2 (ms)	Test 3 (ms)	Average (ms)
1	19	21	22	21
10	27	28	31	29
50	295	313	317	308
100	908	918	921	916
250	2418	2491	2514	2474
500	5423	5792	5942	5719
1000	12134	11234	11457	11608

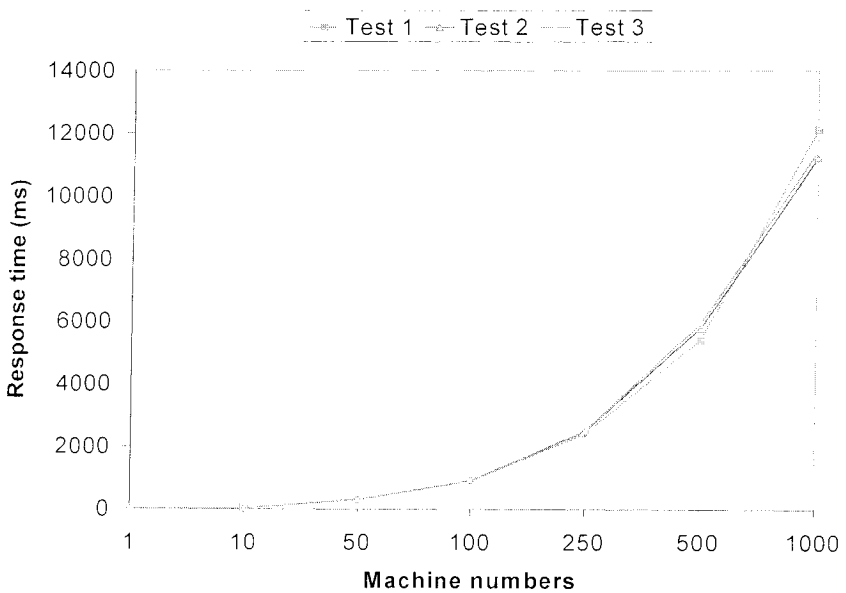

Figure 5.5: Scalability test in a LAN

Another scalability testing was on PlanetLab network, in the Planet Lab at McGill, three nodes in PlanetLab overlay network were chosen to be the servers for scalability test, they are: *node-1.mcgillplanetlab.org*, *nodea.howard.edu*, and *planetlab1.cs.dartmouth.edu*. In each of the server machines, a bunch of XML-RPC server instances were initiated before the client sending out requests, each instance is matching to a specific port number to mimic an independent machine in a real Galaxy system. The total requests were evenly distributed into three servers where each server had about one third of total server instances to accept requests. The client invoked remote procedure calls on those XML-RPC server instances running on the three servers simultaneously.

Table 5.3 shows the different response times for executing broadcasting commands to remote machines. Three groups of tests were executed and it shows that the response times were quite small when the number of nodes was below one hundred which is quite acceptable in real situation. When number of nodes increased to 500, the response time was little longer at around 7.5 seconds. It is a little bit longer but still can be categorized to acceptable level. It needs more than 40 seconds to broadcast to 1000 machines, this is quite long for normal users' tolerance, this suggests that current Shell design will perform very well when simultaneously communicating with up to 100 machines, and can perform in an acceptable level when the broadcasting nodes up to 500. Further than that, a long delay is expected.

Table 5.3: Response time of broadcasting in PlanetLab network

Machine numbers	Test 1 (ms)	Test 2 (ms)	Test 3 (ms)	Average (ms)
1	71	85	76	77
10	101	112	119	111
100	656	988	712	785
500	7506	7403	7866	7592
1000	45556	48526	41256	45113

(Tested at Planet Lab at McGill University, three Linux nodes in the PlanetLab overlay network were chosen to be servers, the client is a Linux machine in the McGill computer lab which has access to internet. The machine numbers are the number of XML-RPC server instances instantiated in the three server nodes, these instances were evenly distributed among three server nodes. The three nodes are: node-1.mcgillplanetlab.org, nodea.howard.edu, and planetlab1.cs.dartmouth.edu, which are physically located at McGill, Howard and Dartmoutn Universities)

Figure 5.6 shows the response time increase trend as the number of nodes are increased, it can be seen that when the machine number is less than 100, the response time increase is quite slow, which demonstrated a good scalability, and when nodes number was up to 500, the delay increase is a little bit higher, after the 500 threshold, the response time increase jumped quickly.

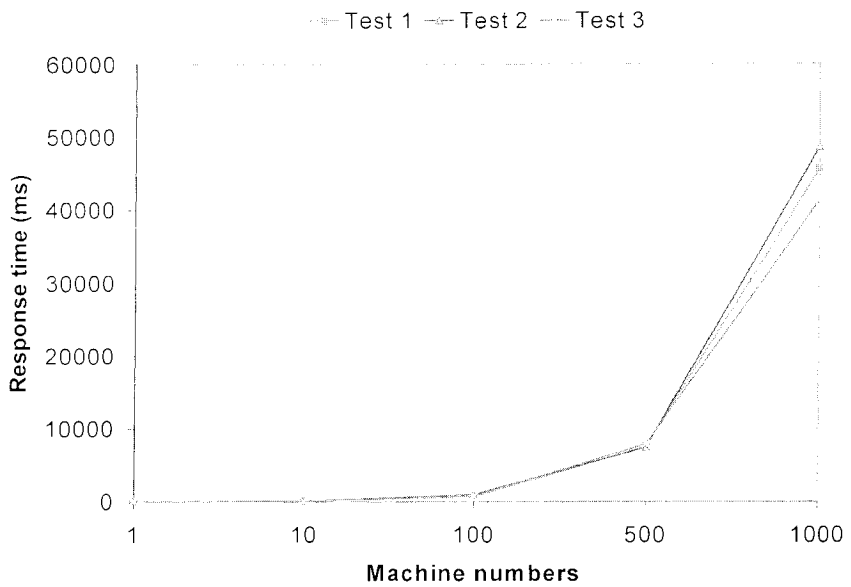

Figure 5.6: Scalability test in PlanetLab network

Although the full Galaxy system will hold thousands or millions of nodes, according to its RAN organization, the Galaxy shell will only directly communicate commands to the "cluster" that is allocated for a particular invocation. That is a shell won't be required to communicate with all the resources that are participating in the Galaxy system. There will be correspondingly large number of invocations of the Galaxy shells that are communicating with the nodes.

It needs to be pointed out that the scalability test is a based on an academic network, which may not be able to reflect the complexity of the network in the real world. Since all machines in PlanetLab are Linux machines with Redhat OS, which may not be able to represent the real situation where Galaxy system may encounter. The bandwidth between nodes in PlanetLab is quite broad due to the advantage of academic institutes' network system, while in the real world: the network connection may vary very much from a telephone dialup to a high speed T1 cable. Although there is some limitation, this test does show us some information about the scalability of the Galaxy Shell, which is the threshold of parallel remote procedure calls. As Galaxy system matures, more tests could be executed based on more "real" situations.

CHAPTER 6

CONCLUSION

Galaxy is a project that implements a public computing utility. It is used to explore solutions for many problems commonly existing in a PCU system, such as scalability, fault tolerance, security management, and resource management. Galaxy combines both P2P and Grid computing ideas. Galaxy neutralizes the resource providers and simplifies the resource collecting, organizing and leasing work. Galaxy differs itself from other public computing utility by its core commoditization concept, relaxed participation models and geographically scalable resource management architecture.

Galaxy shell is at the service layer of Galaxy system, talks downward to the middleware level and provides upwards an interface for the application level. Galaxy shell is a module in Galaxy service layer provides providing searching, binding, allocating and consuming resources in the resource pool. It eases the Galaxy research on sub layer organization. In a big picture, Galaxy shell is composed by three modules, which are shell, local daemon and peer kernel. Shell is responsible for sending requests either locally or remotely in the resource pool, local daemon is running on each node inside Galaxy and accepting and processing commands sent from shell, either locally or remotely; peer kernel is responsible for communicating with outside Galaxy nodes, reporting and collecting information to and from GRMS.

Galaxy shell uses JavaCC to parse input commands to provide a clean, maintainable and extensible syntax checking mechanism. XML-RPC protocol is used for remote communication among distributed Galaxy nodes. Galaxy shell supports commands, which can be used to search, bind resources according to both static and dynamic constraints; multicast and broadcasting are also provided for supporting Galaxy GRMS research. Galaxy shell is small, which means it is easy to be download from Internet for a machine wanting to join Galaxy and get the Galaxy Daemon. Installing and using Galaxy shell is also straightforward as its commands are designed to be similar to the normal Unix Shell we use everyday. From the tests executed at the Planet Lab of McGill University, it can be seen that Galaxy shell performs very well for both internal and external commands; the response times are quite fast and are acceptable for human interactions, from a human being's view. Galaxy shell also demonstrates very good scalability in accordance

with the current Galaxy design goals. Thus it is suitable for supporting Galaxy research in the future as well. Galaxy shell is also easy to be extended, the shell, parser, local daemon and peer kernel modules provide a clear-cut organization which make it very easy to read, maintain, change and extend.

Galaxy research is still going on, the core module of Galaxy, GRMS is still in conceptual design phase, current Galaxy shell is designed and implemented based on current knowledge of GRMS, thus the local daemon and peer kernel modules are far from mature. However, Galaxy shell creates a start for the future extension and revision. The basic framework is there and it is ready for future Galaxy researchers to extend and revise.

Possible future research in Galaxy shell could be:

- Explore the local daemon design and implementation, including authentication and sandbox, which should be integrated into the local daemon module of Galaxy shell
- Further explore the remote procedure call process, make it more stable, reliable and efficient, local daemon should be able to guarantee a resource reservation, a process suspension and resume control system is deserved to be explored
- Explore the peel kernel operation mechanism as the GRMS research goes deeper.

REFERENCES

[1] Advanced Networking Research Lab, *Galaxy Developers' Journal*, www.cs.mcgill.ca/~anrl/PROJECTS/devel_journal.html

[2] Balasubramaneyam Maniymaran (2004), *RAN Naming and Discovery*, Department of Electrical and Computer Engineering, McGill University, Montreal, QC, Canada, 2004

[3] Andrew W. Appel and Jens Palsberg, *Modern Compiler Implementation in Java, Second Edition*, Cambridge University Press © 2002

[4] Ronald Mak, *Writing Compilers and Interpreters*, Second Edition, Wiley Computer Publishing, 1996

[5] D.P. Ghormley, D. Petrou and S.H. Rodrigues, *GLUnix: a Global Layer Unix for a Network of Workstations*, Software Practice and Experience, Vol 28(9), 1998:929-961, http://now.cs.berkeley.edu/Glunix/glunix.html

[6] C.M Tan, C.P. Tan and W.F. Wong, *Shell over a Cluster (SHOC): Towards Achieving Single System Image via the Shell*, Department of Computer Science, National University of Singapore

[7] Allamaraju, Subrahmanyam et al, *Professional Java Server programming: J2EE edition*, Wrox Press, 2000

[8] Project JXTA, *www.jxta.org*.

[9] Project Apache XML-RPC, *http://ws.apache.org/xmlrpc/*

[10] Project JShell, *http://homepage.mac.com/pcbeard/JShell/*

[11] Project Java Compiler Compiler (JavaCC), *https://javacc.dev.java.net/*

[12] Wrox Press, *Making P2P interoperable: The Jxta command shell*, Sept 2001

[13] S. Zhou, "LSF: Load Sharing in Large-scale Heterogeneous Distributed Systems." Proc. Of Workshop of Cluster Computing, Dec 1992

[14] R. Dornfest, *Learning the JXTA Shell*. http://www.openp2p.com/pub/app2p/2001/04/25/learning_jxta_shell.html

[15] S. Oaks, B. Traversat, L. Gong, *JXTA in a Nutshell*, O'Reilly Press, 0-596-00236-X, Sept. 2002.

[16] Project JXTA Protocols Specification, *http://spec.jxta.org/v1.0/docbook/JXTAProtocols.pdf*.

[17] Bernard Traversat, Ahkil Arora, Mohamed Abdelaziz, Mike Duigou, Carl Haywood, Jean-Christophe Hugly, Eric Pouyoul, Bill Yeager, *Project JXTA 2.0 Super-Peer Virtual Network*,

[18] V. Sunderam, *PVM: A Framework for Parallel Distributed Computing*, Concurrency: pratice and Experience, 2(4):315-539, Dec 1990.

[19] M. Baker, *Cluster Computing White Paper*, http://www.dcs.port.ac.uk/~mab/tfcc/WhitePaper/

[20] Andrew Birrell and Bruce Nelson. *Implementing remote procedure calls*. ACM Trans. Computer Systems, 2(1): 39–59, February 1984

[21] Ma, M., Wang, C., and Lau, F. Forthcoming. JESSICA: *Java-enabled single-system-image computing architecture*. Journal of Parallel and Distributed Computing

[22] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, *A Scalable Content Addressable Network*, ACM SIGCOM, 2001.

[23] F. Dabek, E. Brunskill, M.F. Kaashoek, D. Karger, R. Morris, I. Stoica, and H. Balakrishnan, *Building Peer-to-Peer Systems with Chord, a Distributed Lookup Service*, 2001.

[24] Thomas E. Anderson, David E. Culler, David A. Patterson, and the NOW Team. *A Case for NOW (Networks of Workstations)*. IEEE Micro, February 1995.

[25] Andrea C. Dusseau, Remzi H. Arpaci, and David E. Culler. *Effective Distributed Scheduling of Parallel Workloads*. In Proceedings of the 1996 ACM SIGMETRICS Conference, 1996.

[26] Alan Mainwaring and David Culler. *Active Message Applications Programming Interface and Communication Subsystem Organization*. Technical Report CSD-96-918, University of California at Berkeley, October 1996.

[27] V. Sunderam. PVM: *A Framework for Parallel Distributed Computing. Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[28] Songnian Zhou. LSF: load sharing in large-scale heterogeneous distributed systems. In *Proceedings of the Workshop on Cluster Computing*, December 1992.

[29] M. E. Lesk and E. Schmidt, *Lex - A Lexical Analyzer Generator*, http://www.cs.utexas.edu/users/novak/lexpaper.htm

[30] How lexers work, http://www.cs.man.ac.uk/~pjj/cs2121/ho/node6.html

[31] The JavaCC FAQ, http://www.engr.mun.ca/~theo/JavaCC-FAQ

[32] Java RMI & CORBA, a comparison of two competing technologies, http://www.javacoffeebreak.com/articles/rmi_corba

APPENDIX A

PARSER SEED FILE

Following is the file Parser.jj which is the seed file being used to produce the lexical analyzer and parser of Galaxy Shell, this file is located at Package *gshell.parser*.

```
options {
  LOOKAHEAD = 1;
  CHOICE_AMBIGUITY_CHECK = 2;
  OTHER_AMBIGUITY_CHECK = 1;
  STATIC = false;
  DEBUG_PARSER = false;
  DEBUG_LOOKAHEAD = false;
  DEBUG_TOKEN_MANAGER = false;
  ERROR_REPORTING = true;
  JAVA_UNICODE_ESCAPE = false;
  UNICODE_INPUT = false;
  IGNORE_CASE = false;
  USER_TOKEN_MANAGER = false;
  USER_CHAR_STREAM = false;
  BUILD_PARSER = true;
  BUILD_TOKEN_MANAGER = true;
  SANITY_CHECK = true;
  FORCE_LA_CHECK = false;
}

PARSER_BEGIN(Parser)

package gshell.parser;
```

```
import java.util.*;
import gshell.*;

public class Parser {

  public static void main(String args[]) throws ParseException {
    Parser parser = new Parser(System.in);
    parser.Process();
  }
}

PARSER_END(Parser)

void Process() :
{
  Token t;
  // following are temporary vars.
  String built_in_command = "";
  String external_command = ""; // including "ls", "cd", and other executable file names
  String destination = "";  // rename "domain" to "destination" for avoiding naming conflict
  String attr_constraint = "";
  String type_constraint = "";
  String dynamic_constraint = ""; // for string connection
  String other_parameter = ""; // for string connection
  String node_range_constraint = "";
 int node_number_constraint = 1;
}
{
 (
  t = <domain>
          {
           destination = t.image; // if more than one domain occurs, the last one override previous
one(s)
          }
```

```
|
t = <attrConstraint>
        {
         attr_constraint = t.image;
        }
|
t = <typeConstraint>
        {
         type_constraint = t.image;
        }
|
t = <loadConstraint>
        {
         dynamic_constraint += t.image;
         dynamic_constraint += ",";
        }
|
t = <proximityConstraint>
        {
         dynamic_constraint += t.image;
         dynamic_constraint += ",";
        }
|
t = <speedConstraint>
        {
         dynamic_constraint += t.image;
         dynamic_constraint += ",";
        }
|
t = <qosConstraint>
        {
         dynamic_constraint += t.image;
         dynamic_constraint += ",";
        }
```

```
|
t = <reservationConstraint>
        {
         dynamic_constraint += t.image;
         dynamic_constraint += ",";
        }
|
t = <desiredThroughput>
        {
         dynamic_constraint += t.image;
         dynamic_constraint += ",";
        }
|
t = <commonDashParameter>
        {
         other_parameter += t.image;
         other_parameter += " ";
        }
|
t = <commonWeirdParameter>
    {
     other_parameter += t.image;
     other_parameter += " ";
    }
|
t = <nodeNumberConstraint>
        {
         node_number_constraint = Integer.valueOf(t.image.substring(4)).intValue();
        }
|
t = <wildcard>
    {
     other_parameter += t.image;
```

```
        other_parameter += " ";
      }
  |
  t = <idWithWildcard>
      {
        other_parameter += t.image;
        other_parameter += " ";
      }
  |
  t = <nodeRangeConstraint>
      {
        node_range_constraint += t.image;
      }
  |
  t = <id>
        {
      String s = t.image;
      if ( !(built_in_command == "" && external_command == "") ) { // command already exists
                other_parameter += s;
                other_parameter += " "; // sepearate by white space for convinence
        } else { // no any command yet
                if (s.equalsIgnoreCase("lstype"))
                        built_in_command = "lstype";
                if (s.equalsIgnoreCase("lsbind"))
                        built_in_command = "lsbind";
                if (s.equalsIgnoreCase("bind"))
                        built_in_command = "bind";
                if (s.equalsIgnoreCase("unbind"))
                        built_in_command = "unbind";
                if (s.equalsIgnoreCase("setdomain"))
                        built_in_command = "setdomain";
                if (s.equalsIgnoreCase("getdomain"))
                        built_in_command = "getdomain";
                if (s.equalsIgnoreCase("setnode"))
```

```
                built_in_command = "setnode";
            if (s.equalsIgnoreCase("getnode"))
                built_in_command = "getnode";
            if (s.equalsIgnoreCase("settype"))
                built_in_command = "settype";
            if (s.equalsIgnoreCase("gettype"))
                built_in_command = "gettype";
            if (s.equalsIgnoreCase("setattr"))
                built_in_command = "setattr";
            if (s.equalsIgnoreCase("getattr"))
                built_in_command = "getattr";
            if (s.equalsIgnoreCase("deftype"))
                built_in_command = "deftype";
            if (built_in_command == "")  // not a built-in command
                external_command = s; // now it is an external commmand

        }
        }
    )* ( <endOfCommand> | <EOF> )
        {
        if (dynamic_constraint.length() > 0) {
            dynamic_constraint=dynamic_constraint.substring(0,
dynamic_constraint.length() - 1);
        }

        if (other_parameter.length() > 0) {
            other_parameter = other_parameter.substring(0, other_parameter.length() - 1);
        }

        // return parse result by setting shell parameters
        gshell.built_in_command = built_in_command;
        gshell.attr_constraint = attr_constraint;
        gshell.type_constraint = type_constraint;
        gshell.dynamic_constraint = dynamic_constraint;
```

```
            gshell.node_number_constraint = node_number_constraint;
            gshell.node_range_constraint = node_range_constraint;
            gshell.external_command_list.add(external_command); // always add
            gshell.destination_list.add(destination);
            gshell.other_parameter_list.add(other_parameter);
            }
}


// lexical specification begin here

TOKEN :
{
 < id: ["a"-"z","A"-"Z"] ( ["a"-"z","A"-"Z","0"-"9"] )* >
 |
 < num: (["0"-"9"])+ >
 |
 < validNumOfNodes: ["1"-"9"](<num>)? >
 |
 < relationOp: ("&&"|"||") >
 |
 < logicOp: ("<"|">"|"<="|">="|"="|"<>") >
 |
 < relationPair: <id><logicOp>(<id>|<num>(<id>)?) >
 |
 < attribute: <relationPair>( <relationOp><relationPair> )* >
 |
 < wildcard: ("*"|"?")? >
 |
 < dot: "." >
 |
 < idWithWildcard: (<domainId>)+ >
 |
 < domainId: (<id>|<wildcard>) >
```

```
  |
  < domain: (<domainId>":"<domainId>":"<domainId>|"@"<id>) >
  |
  < commonDashParameter: "-"<id> >
  |
  < commonWeirdParameter: ( "/"<id>|(<id>)?<wildcard>(<id>)?|(<id>)?<dot>(<id>)? ) >
  |
  < attrConstraint: ("--a")(" ")*<attribute> >
  |
  < typeConstraint: ("--t")(" ")*<id> >
  |
  < loadConstraint: ("--l")(" ")*("heavy"|"medium"|"low") >
  |
  < desiredThroughput: ("--dt")(" ")*("heavy"|"medium"|"low")>
  |
  < qosConstraint: ("--q")(" ")*("D"|"d"|"B"|"b")>
  |
  < reservationConstraint: ("--r")(" ")*<validNumOfNodes> >
  |
  < proximityConstraint: ("--p")(" ")*("far"|"medium"|"close") >
  |
  < speedConstraint: ("--s")(" ")*("fast"|"medium"|"slow") >
  |
  < nodeRangeConstraint: <id>"["<validNumOfNodes>"-"<validNumOfNodes>"]" >
  |
  < nodeNumberConstraint: ("--n")(" ")*<validNumOfNodes> >
  |
  < endOfCommand: ("\r"|"\n"|"\r\n") >
}

SKIP :
{ " " | "\t" }
```

APPENDIX B

SOURCE CODE DESCRIPTION

Following is a high-level description of the source code of Galaxy shell. Galaxy shell implementation is composed by three components, which are shell, local daemon and peer kernel, shown as Figure B.1. The *gshell* package contains classes to compose the graphic shell interface, it also has a sub package named *parser*, which contains classes tokenizing and syntax checking input commands from the shell interface. The *localdaemon* package contains classes provide Galaxy services; a developer can add new services into this package by creating new classes. The *peerkernel* package contains classes for supporting kernel service of a Galaxy node, such as profiling network information. Figure B.2 shows the classes in the three sub packages. A detailed explanation of the classes in these packages can be referenced in Chapter 4.

Figure B.1 Galaxy shell implementation architecture

Figure B.2 Classes list in three sub packages

A Java document was created for users' convenience, a snapshot of this document is shown in Figure B.3, where the hierarchy of all packages is shown (partially due to the screen limitation).

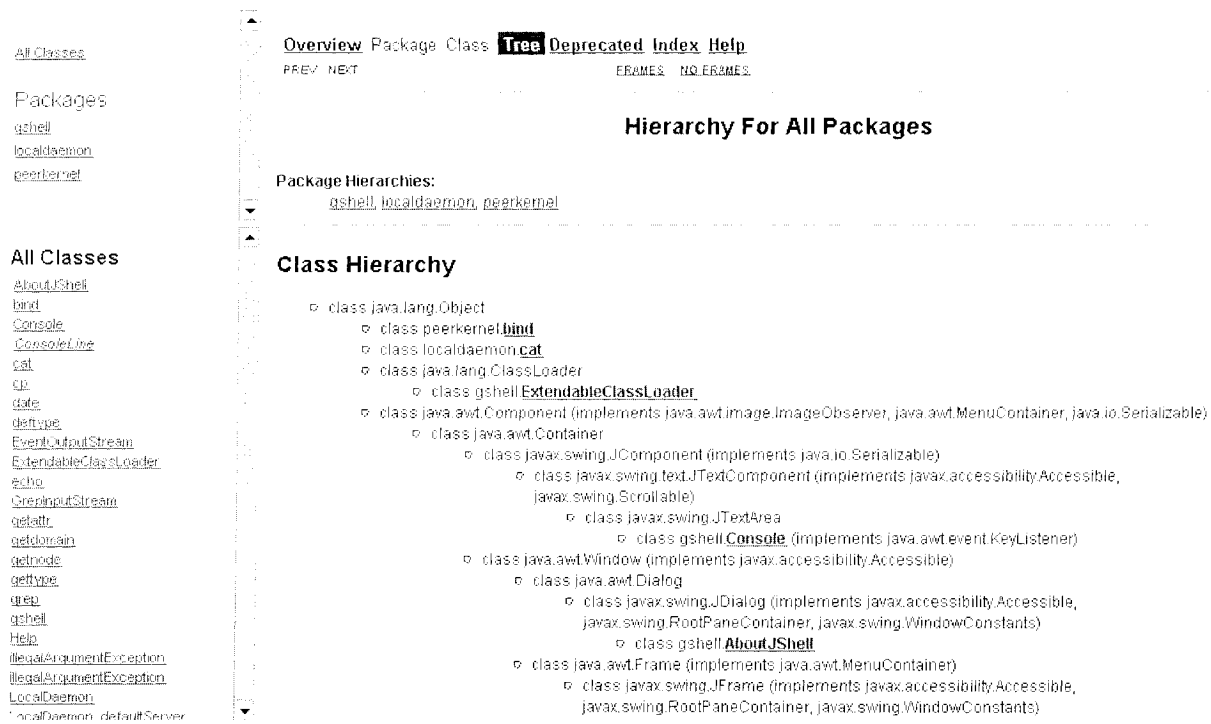

Figure B.3 Snapshot of Java Document of Galaxy shell project

For the detail information for each class or methods in these classes, please refer to Chapter 4 or go through the documentation and click corresponding links.