

California State University, San Bernardino

CSUSB ScholarWorks

Theses Digitization Project

John M. Pfau Library

2002

Spider III: A multi-agent-based distributed computing system

Jianhua Ruan

Han-Shen Yuh

Koping Wang

Follow this and additional works at: <https://scholarworks.lib.csusb.edu/etd-project>



Part of the [Digital Communications and Networking Commons](#)

Recommended Citation

Ruan, Jianhua; Yuh, Han-Shen; and Wang, Koping, "Spider III: A multi-agent-based distributed computing system" (2002). *Theses Digitization Project*. 2249.

<https://scholarworks.lib.csusb.edu/etd-project/2249>

This Project is brought to you for free and open access by the John M. Pfau Library at CSUSB ScholarWorks. It has been accepted for inclusion in Theses Digitization Project by an authorized administrator of CSUSB ScholarWorks. For more information, please contact scholarworks@csusb.edu.

SPIDER III: A MULTI-AGENT-BASED
DISTRIBUTED COMPUTING SYSTEM

A Project
Presented to the
Faculty of
California State University,
San Bernardino

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
in
Computer Science

by
Jianhua Ruan
June 2002

SPIDER III: A MULTI-AGENT-BASED
DISTRIBUTED COMPUTING SYSTEM


A Project
Presented to the
Faculty of
California State University,
San Bernardino

by

Jianhua Ruan

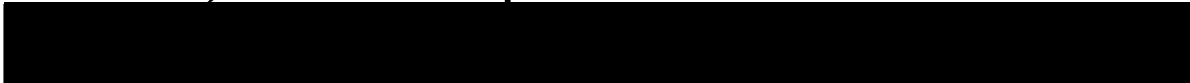
June 2002

Approved by:



Dr. Arturo I. Concepcion, Chair, Computer Science

05 Jun 2002
Date



Dr. Kay Zeroudeh



Dr. Ernesto Gomez

ABSTRACT

This project, Spider III, proposes an architecture and protocol of a multi-agent-based Internet distributed computing system, which provides a convenient development and execution environment for transparent task distribution, load balancing, and fault tolerance. This project presents the design and implementation of a prototype using the Aglets software development kit (ASDK 2.0). The prototype implemented all the core agents, the task distribution protocol, and a set of application programming interfaces, including a simple Task/Slave pattern. A graphical user interface was developed to provide better visual and operational convenience. To validate our design and to test the system performance, a distributed matrix multiplication algorithm was programmed on Spider. Its execution time and distribution efficiency were compared with PVM and sequential programs. The results showed that the Spider system succeeded to utilize resources across multiple LANs and multiple host architectures, and that for appropriately large matrix size the Spider system has better overall performance over the PVM and sequential programs.

ACKNOWLEDGMENTS

Without the support and help of my friends, my colleagues, and my family, this work would not have been possible.

First of all, I own special thanks to my advisor, Dr. Arturo I Concepcion, who is a constant flow of valuable advice. He was always ready to discuss new ideas and to provide very important feedback on all issues and problems that I encountered during my work.

I also thank all my colleagues in the Spider team, especially Rodelyn Samson for fruitful discussion during the initiation of the architecture. Special thanks to Chunyan Ma who proof-read some of the documentation.

Most importantly, I thank my family for all their love, support, and encouragement. My parents and my sister supported me throughout my studies in many ways.

Finally, I want to thank all friends that helped me during my staying at CSUSB. Zhuo, Xiwei, Li, Hao and Yan have always treated me as a family, and offered me numerous helps during the stressful time of my life.

The support of the National Science Foundation under the award 9810708 is gratefully acknowledged.

TABLE OF CONTENTS

ABSTRACT	iii
ACKNOWLEDGMENTS	iv
LIST OF TABLES	ix
LIST OF FIGURES	x
CHAPTER ONE: INTRODUCTION	
1.1 Background	1
1.2 Purpose of the Project	2
1.3 Limitations of the Project	3
1.4 Organization of the Documentation	4
Chapter Two: Software Requirements Specification	
2.1 Introduction	5
2.1.1 Scope	5
2.1.2 Overview	6
2.2 Overall Description	7
2.2.1 Product Perspective	9
2.2.2 Product Functions	15
2.2.3 User Characteristics	19
2.2.4 Constraints	19
2.2.5 Assumptions and Dependencies	19
2.3 Specific Requirements	20
2.3.1 External Interface Requirements	20
2.3.2 Functional Requirements	24

2.3.3	Performance Requirements	25
2.3.4	Software System Attributes	25
CHAPTER THREE: DESIGN OF THE SPIDER III SYSTEM		
3.1	System Design	27
3.1.1	Core Agents in the Spider System	27
3.1.2	The Task Distribution Protocol	30
3.1.3	Load Balancing	33
3.1.4	Fault Tolerance	34
3.2	Architecture	36
3.2.1	Package edu.csusb.spider.system	39
3.2.2	Package edu.csusb.spider.finder	41
3.2.3	Package edu.csusb.spider.launcher	42
3.2.4	Package edu.csusb.spider.misc	44
3.2.5	Package edu.csusb.spider.pattern	45
3.2.6	Package edu.csusb.spider.group	46
3.3	Detailed Design	47
3.3.1	Synchronizer	48
3.3.2	Registry	50
3.3.3	Multicaster	51
3.3.4	UsageMonitor	51
3.3.5	SystemVerifier	52
CHAPTER FOUR: SPIDER USER APIS		
4.1	Task Creation	56
4.2	Message Passing	56

4.3	Group Functions	58
4.4	Asynchronous Tasks	58
4.5	Program Examples	59
CHAPTER FIVE: SYSTEM TESTING AND PERFORMANCE ANALYSIS		
5.1	Testing Environments	61
5.2	Testing Methods	61
5.3	Testing Results	62
5.4	Performance Analysis	66
5.4.1	Execution Costs	66
5.4.2	Synchronization Costs	68
CHAPTER SIX: MAINTENANCE MANUAL		
6.1	Obtaining a Copy	71
6.2	Directory Organization	71
6.3	System Requirements	72
6.4	Installation	72
6.5	Configuration	73
6.6	Running the Spider System	73
6.6.1	Startup Options	74
6.6.2	Commands of Spiderd Console	74
6.6.3	Tools of Xspider Interface	76
6.7	Developing Applications on Spider III	77
6.7.1	Writing Source Program Using Spider APIs	77

6.7.2	Compilation and Deployment	79
6.7.3	Running the Application	79
CHAPTER SEVEN: CONCLUSIONS AND FUTURE DIRECTIONS		
7.1	Conclusions	81
7.2	Future Directions	84
APPENDIX A: GLOSSARY		87
BIBLIOGRAPHY		90

LIST OF TABLES

Table 5.1. Performance of Distributed Matrix Multiplication	64
--	----

LIST OF FIGURES

Figure 2.1. Deployment Diagram for Spider III	12
Figure 2.2. Spider III System Functionalities	16
Figure 2.3. Spider III Graphical User Interface Main Frame	21
Figure 2.4. Load Task Dialog	22
Figure 3.1. Components of Spider Host	29
Figure 3.2. An Example of How an Application is Distributed to the Spider System	32
Figure 3.3. Overview of the Spider III Class Diagram ...	38
Figure 3.4. Class Diagram of the Package <i>edu.csusb.spider.system</i>	40
Figure 3.5. Class Diagram of the Package <i>edu.csusb.spider.finder</i>	42
Figure 3.6. Class Diagram of the Package <i>edu.csusb.spider.launcher</i>	44
Figure 3.7. Class Diagram of the Package <i>edu.csusb.spider.misc</i>	46
Figure 3.8. Class Diagram of the Package <i>edu.csusb.spider.group</i>	47
Figure 3.9. Pseudo Code of Synchronizer	49
Figure 3.10. Pseudo Code of Registry	50
Figure 3.11. Pseudo Code of Multicaster	51
Figure 3.12. Pseudo Code of UsageMonitor	52
Figure 3.13. Pseudo Code of SystemVerifier	53
Figure 4.1. Task Interface	55

Figure 4.2. Program Example	60
Figure 5.1. Sample Screen of Running the Distributed Matrix Multiplication Program	62
Figure 5.2. Comparisons Between Performance of Spider and PVM	65
Figure 6.1. Xspider User Interface	76
Figure 6.2. HelloWorld.java	78
Figure 6.3. Output of HelloWorld.java	80

CHAPTER ONE

INTRODUCTION

1.1 Background

Spider is an on going distributed computing project in the Department of Computer Science, California State University San Bernardino (CSUSB). It was first proposed as an object-oriented distributed system by Han-Sheng Yuh in his Master's thesis in 1997 [2]. And it has been thereafter further developed by Koping Wang in his Master's Project [3], where he made large contribution and implemented the Spider II system. Spider I & II systems are developed using C++, that runs on Unix or Unix-like platform.

Spider I system consists of four major components: Task Manager, Registry Server, Object Server Broker, and Object-Servers. Task Manager keeps track of all task activities. Registry Server auto-detect Object-Servers and send the list of available servers to the Task Manager. An Object Server Broker is the node where a user initiates her tasks. It connects to the Task Manager to get available computers and distribute task to other Object Servers. An Object Server is a computer that is requested to join in the distributed computation. Mirrors for both Task Manager

and Registry Server were introduced in the Spider II system. A failure of the task manager or registry server will activate its mirror and all further communication is then redirected to the mirror.

However, Spider I & II systems, as well as most other traditional distributed computing system, such as PVM and MPI [15], have several limitations. First, it does not allow heterogeneous host architecture and operating system to join in the computation. Although PVM allows different architectures to coexist in the system, it does require applications to be compiled for each type of architecture separately. Second, before starting a distributed application, executable files have to be manually transferred to destination hosts in advance if a network file system is not in use, which significantly reduces efficiency. Third, the user of the system needs to have remote login permission to the destination host in order to perform computation, which opens serious security breach, especially for computers connected to the Internet.

1.2 Purpose of the Project

This project, Spider III, attempts to propose and implement a distributed computing environment for developing

and executing large scale distributed applications. It will provide partial solutions to the problems of traditional distributed computing systems discussed above, using agent-oriented programming model.

In the Spider III system, system functions, such as monitoring hosts, managing tasks, and inter-process communication, are carried out by agents. User tasks are also done by agents, which carry algorithms and data and are capable of migrating to remote hosts to execute, either due to response to commands explicitly issued by users, or implicitly decided by the system. The Spider III system will be able to utilize free CPU cycles of any computer on the Internet, given that a daemon is running on the physical node. The execution will not be limited by the number of nodes in the network, the type of host architecture, and the operating system running on each individual node.

1.3 Limitations of the Project

This project will not handle security mechanisms of the system. Thus, in this project we assume that the owner of each node is willing to install and run the program as a daemon process. The security mechanism is another project

and is beyond the scope of this project. However, this project does provide a minimum level of security: all remote tasks, except the system agents, will not be able to directly operate on any files.

1.4 Organization of the Documentation

The remaining sections of this documentation will be organized as follows: Chapter 2 describes the software requirement specification. Chapter 3 provides a detailed description of the system architecture and design. Chapter 4 introduces the Spider III user APIs and provides a sample program of programming on Spider. Chapter 5 is the system test and analysis. Chapter 6 is the maintenance manual. Finally, Chapter 7 concludes the project and lists suggestions for future developments.

CHAPTER TWO
SOFTWARE REQUIREMENTS
SPECIFICATION

2.1 Introduction

This software requirement specification is for the Spider III - an agent-based Internet Virtual Machine - project. This will be the master's project of Jianhua Ruan for the requirement of Master of Science Degree in the Department of Computer Science, CSUSB.

2.1.1 Scope

Spider is an on going distributed computing project in Department of Computer Science, California State University San Bernardino (CSUSB). It was first proposed as an object-oriented distributed system by Han-Sheng Yuh in his Master's thesis in 1997 [2]. And it has been thereafter further developed by Koping Wang in his Master's Project, where he made large contributions and implemented the Spider II system [3].

This project is based on the Spider II system. However, a new computational model is conceived, and the system will be implemented in agent-based approach. Thus it

is named Spider III to be differentiated from previous versions.

In the Spider III system, system functions, such as monitoring hosts, managing tasks, and inter-process communication, are carried out by agents. User tasks are also done by agents, for which users implement their own algorithms. User agents can migrate to another node, either due to response to commands explicitly issued by users, or implicitly decided by the system. Spider III system is proposed as the Internet computing system thus it is able to scale to very large number of nodes. Because of the unpredicted properties of the Internet, Spider III system possess the ability to handle heterogeneous architecture and operating system, and must provide good fault tolerance.

2.1.2 Overview

The rest of the software requirement specification organizes as follows. Section 2 of this chapter follows the guidelines of Std 830-1998 IEEE recommended Practice of Software Requirements Specifications [1]. This section provides product perspective, a summary of product functions, a description of the characteristics of the expected users, a listing of development constrains, a list

of assumptions and dependencies. Section 3 of this chapter presents the specific requirements. This section presents external interface requirements, functional requirements, performance requirements, design constraints, and software system attributes. Functional requirements are organized by objects and comply with the template of Std 830-1998 IEEE Appendix A.4.

2.2 Overall Description

Spider is the name of the distributed computing system developed in the Department of Computer Science, California State University, San Bernardino. The first version of Spider system (Spider I) was designed and developed by Han Sheng Yuh in his Master's thesis in 1997. Spider I is developed using C++, to be run on Unix or Unix-like platform. It was tested on SGI Indigo workstations and Irix operating system. RPC (remote procedure call) and BSD socket interface were used for implementation. TCP and UDP protocols are employed to suit for the different needs in implementation.

The second version of the Spider system is developed by Koping Wang in 1999. The Spider II system optimized performance by reducing the communication protocol

complexity and incorporating load-balancing service. It also stabilized the multi-tasking operations and achieved fault tolerance by having mirrors for both Task Manager and Registry Server. Ease of adding new services is another improvement. Spider II runs on top of the Unix or Linux operating system. Wang implemented the following distributed computing services: distributed quick sort, distributed prime number search, and distributed matrix multiplication to run on Spider II system. A paper about Spider II has been cited and published in PDCS (Parallel and Distributed Computing System) 2000.

Spider I & II systems consist of four major components: Task Manager, Registry Server, Object Server Broker and Object-Servers. Task Manager keeps track of all task activities. Registry Server auto-detect Object-Servers and send the list of available servers to the Task Manager. An Object Server Broker is the node where user initiates her task. It connects with Task Manager to get available computers and distribute tasks to other Object Servers. An Object Server is a computer that is requested to join in the distributed computation.

Besides the components in Spider I system, Spider II system introduced mirrors for both Task Manager and

Registry Server, because of their important roles in the distributed computing. A failure of the task manager or registry server will activate its mirror and all further communication is then redirected to the mirror.

2.2.1 Product Perspective

The Spider III system will be agent-based, and will be implemented using Java 2. To avoid developing agent architecture from scratch, the Spider III system uses Aglets 2.0 as agent architecture. The Spider III system will provide Java GUI based user interface. The software interface requirement is that it must have Java 1.2 or greater runtime environment or version installed, the communication interface requires support for TCP and UDP protocols. The operation requirements are that a user must be able to initiate a task from wherever a Spider system is installed, including mobile devices where network connections are temporary.

2.2.1.1 System Interfaces. In the Spider system, a user task will be conducted in the following steps,

- (1) User initiates a task via the user interface and an agent that is going to execute is loaded into the Agent Server;

- (2) Agent Server requests local resource manager for available computation resources;
- (3) Agent is marshaled and dispatched to remote host;
- (4) Agent is unpacked and starts execution at remote host. It may communicate with agents at other remote hosts if necessary;
- (5) Agent finishes its task and return to its origin host, present result to user.

Alternative 1: user disconnects after initiates a task

- (5a) Agent finishes its task, and then it is serialized to be stored on file system
- (5b) When its owner connects to the system again, the agent is activated and returned to its original host, present result to user.

Alternative 2: user aborts a task before it finishes, or unexpected error happens

- (5c) Original host request the agent server about location of subagent belongs to this task, send message to let remote host kill those subagents.

The resource manager of the Spider system contains a mobile agent, which migrates continuously from node to node to collect information about available computation resources. The agent manager contains information about all

agent initiated from the local host, including their status and their location.

Figure 2.1 is the deployment diagram that shows the physical layout of components on hardware nodes.

2.2.1.2 User Interfaces. User interfaces for Spider III will be designed in Java Frames (stand alone Java applications). The following features will be incorporated to produce a more descriptive representation of the interface.

2.2.1.2.1 Main Frame. The first interface after a user starts the system is the main frame, which provides access to all functions of the Spider III system. It contains a menu bar, a toolbar and four view areas: task view, communication view, agent view and network view respectively.

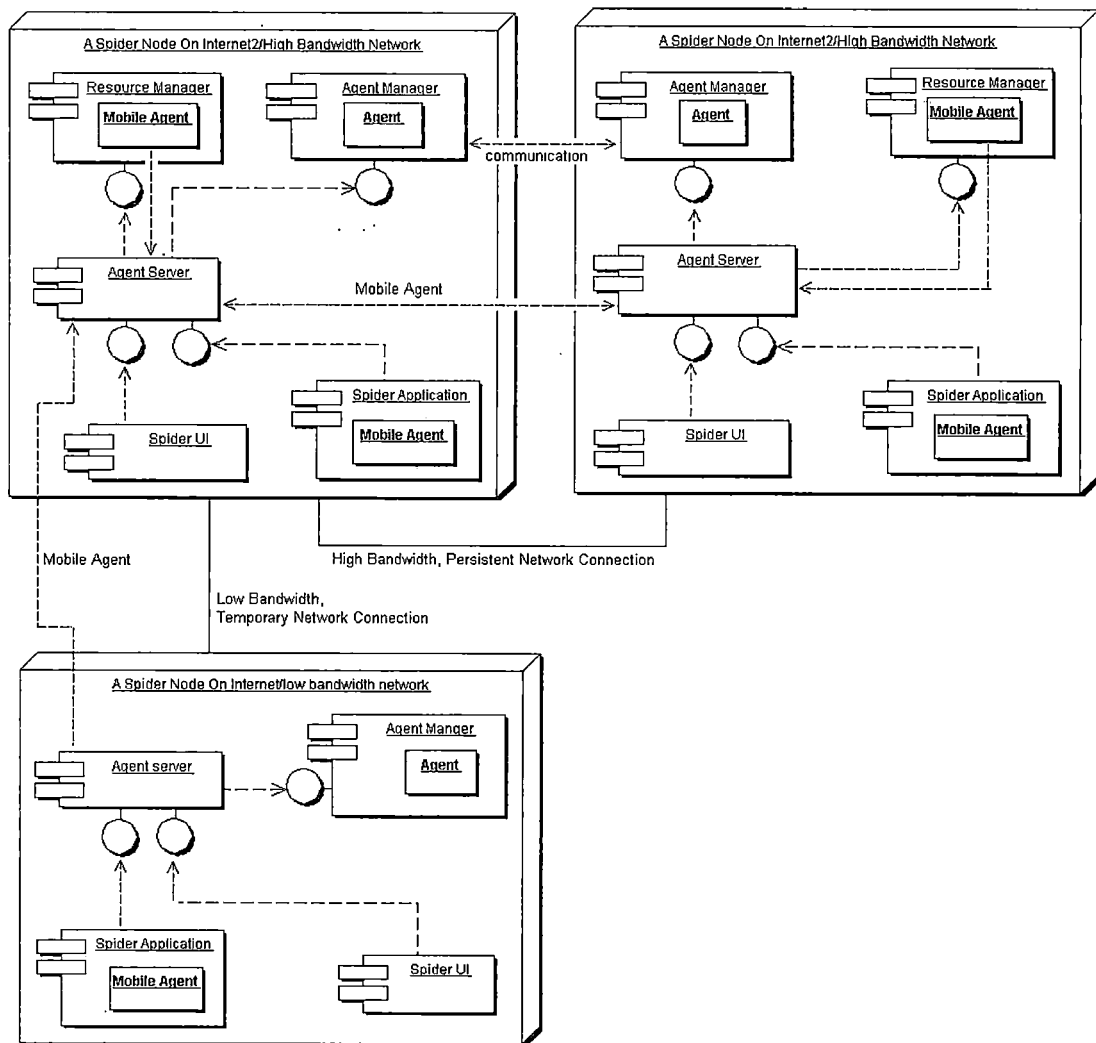


Figure 2.1. Deployment Diagram for Spider III

2.2.1.2.2 Menu Bar. The menu bar on the top of the main frame provides an access to the functions of the Spider III system. There are five menus provided: nodes, tasks, view, control and help, each has their own submenus.

Nodes - this menu is used for management of nodes. It contains submenu *Node Manager*, click on which shall prompt the nodes manager frame.

Tasks - this menu provides functions of task management. It includes four submenus: load, start, dispatch and stop, whose functions are load a task into the system, start a task, send a task to be run remotely, or stop a task, respectively.

View - from the view menu a user can open several other frames to view the system. The Network submenu gives information about machines that are currently utilized by the Spider III system. The Agents submenu shows progress of all agents running on the system. The Event submenu displays agent events.

System - users can change the settings of the system, such as auto connection upon startup, type of network (permanent connection vs. temporary connection), default machine to connect, etc. Another function that could be performed through control menu is disconnect, which is generally used by mobile users or other low speed network users to disconnect from the system after submitting a task to nodes in high speed backbone network.

Help - includes software version information, supporting site, and FAQs.

2.2.1.2.3 Toolbar. Corresponding to each menu item, there is a control in the toolbar for convenience.

2.2.1.3 Hardware Interfaces. Essentially all hardware interfaces will be provided by the operating system.

2.2.1.4 Software Interfaces. Software interfaces are provided in Java 1.3 APIs and ASKD 2.0 APIs. Java is a trademark of Sun Microsystems and ASKD is under IBM public license.

2.2.1.5 Communication Interfaces. The communication interfaces are provided by the underlying Agent Server, which is Aglet in this implementation. It supports both synchronous and asynchronous message, and supports acknowledge-type replies. It uses the Agent Transfer Protocol (ATP) as the mechanism for transferring the serialized data of an agent to its destination, tracking and managing agents.

2.2.1.6 Memory Constraints. At least 32 megabytes of physical RAM is required to run graphical based Java application. 48 megabytes or more are recommended to run large programs for adequate performance.

2.2.1.7 Operations. Users need to install Spider III on their own machines. User initiates their task by interaction to the GUI of the Spider III system. To spawn more nodes automatically, IP addresses of a list of available machines should be stored in the configuration file. Users can also manually add machines into the system during the run time. The task that a user is going to initiate should have resided on the local machine, or he could provide the URL of the code base for a remote task.

2.2.2 Product Functions

Figure 2.2 shows Use Case diagram that graphically depicts the users and principal functions of the Spider III system.

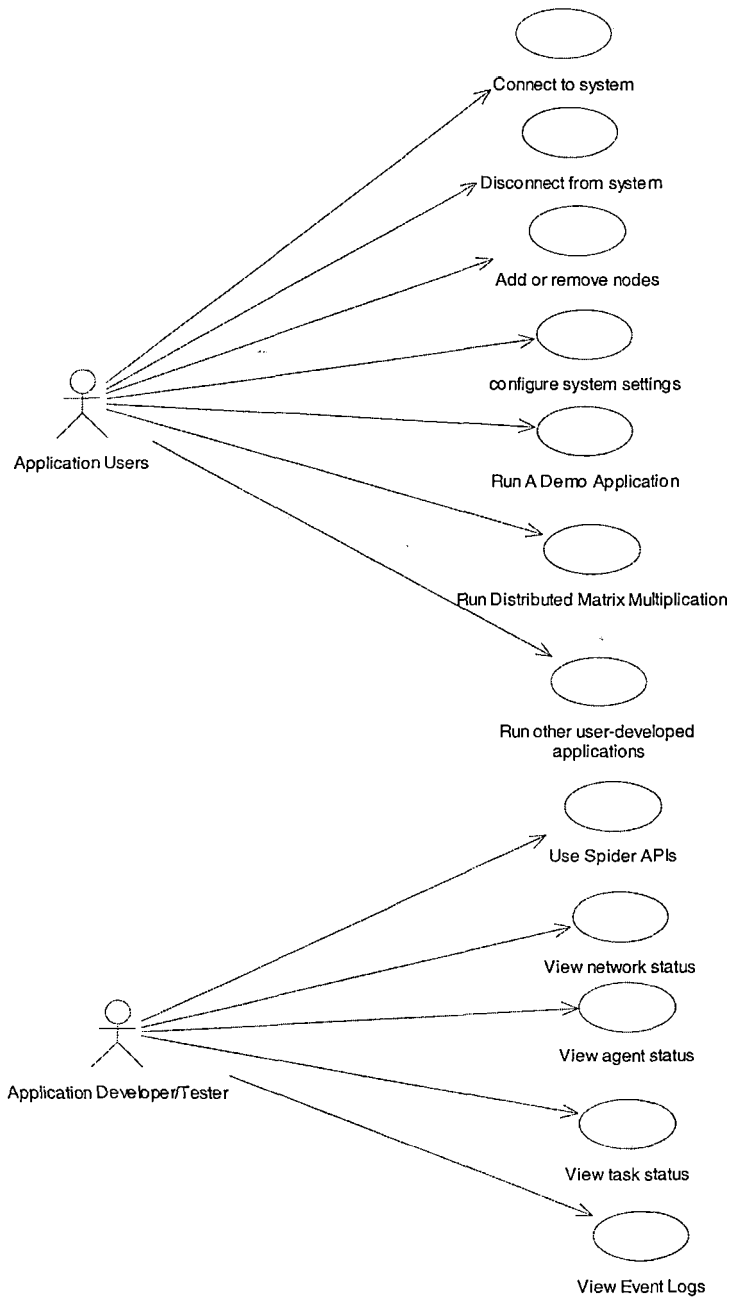


Figure 2.2. Spider III System Functionalities

2.2.2.1 Connect and Disconnect. User can connect and disconnect to the system dynamically.

2.2.2.2 Nodes Addition And Removal. The Spider III System is responsible to collect machines and add them into the system. It can be done automatically at startup time by providing a list of IP addresses, or manually at system run time. A node can also be removed from the system. The Spider III system will provide APIs to application developer so that nodes can be dynamically added or removed during task run time.

2.2.2.3 Task Load, Start, and Dispatch. An agent needs to be loaded into the system before execution. When the Start command is issued, the Spider III system begins to initiate task of the agent. If the Dispatch command is issued, the Spider III system will attempt to find a remote node to run this agent, rather than run it locally. The Spider III system provides APIs so that an application developer can write code to spawn a number of slave agents from the master agent. The Spider III system will automatically allocate these slave agents to available nodes in the system.

2.2.2.4 Run a Demo Application. The system shall provide several demo applications, including distributed

matrix multiplication and other program samples to demo how to use the Spider III APIs.

2.2.2.5 Disposal. The user may choose to abort an unfinished task, no matter it is running locally or remotely. If it is running remotely, the underlying Spider III system will send a message to the agent server where the agent resides, and kill the agent.

2.2.2.6 Use Spider APIs. The Spider III system shall provide a set of APIs that users can use to easily develop their own applications to be run on the Spider III system.

2.2.2.7 View Task Status. The task status viewer shows all tasks information initiated from the local computer, such as task ID, when the task is initiated, how many machines are involved, and how many agents have been dispatched on behalf of this task.

2.2.2.8 View System Agents. The agent status viewer shows all system agents running on the computer. Information displayed includes agent name, ID, and status (active, inactive).

2.2.2.9 View Network Status. The network status viewer shows the machines that are involved in the system.

2.2.2.10 View Event Logs. This function opens up another window to display event logs, such as system

startup, nodes addition or removal, agent creation, agent dispatching, and agent returning.

2.2.3 User Characteristics

The following capabilities are assumed for the various users:

2.2.3.1 Application User. A Spider III application user needs to know how to interact with GUI components such as menus and buttons. They should be able to follow the manual written in plain English.

2.2.3.2 Application Developer. An application developer of Spider III should know programming in Java language and should be familiar with the concepts of distributed computing. It is also assumed that they understand the agent-oriented programming model.

2.2.4 Constraints

The Spider III system has the following constraints: After a running agent migrates to another node, it cannot restart from where it stops. Generally it will run from the beginning, unless an application developer explicitly programming it to be restarted from another entry point.

2.2.5 Assumptions and Dependencies

The Spider III system shall be developed using Aglets Software Development Kit 2.0 (ASDK 2.0) and Java

Development Kit 1.3. It may not be compatible with previous and later versions of Spider and ASDK.

2.3 Specific Requirements

This section of the SRS contains all the software requirements to a level of detail sufficient to enable designers to design the Spider system in conformance with the requirements of this Specification Requirement document. This level of sophistication will also enable testers to generate tests for the system, to verify that it meets the requirements. Every stated requirement will be externally perceivable by users through the usage of sample screen dumps. The requirements include a description of every input, every output, and all functions performed by the system to generate output in response to input.

2.3.1 External Interface Requirements

2.3.1.1 Main Frame. After a user starts the system, they shall see the main frame (see Figure 2.3), from which they interact with the system.

2.3.1.2 Menu Items.

System - contains submenus connect, disconnect, and exit.

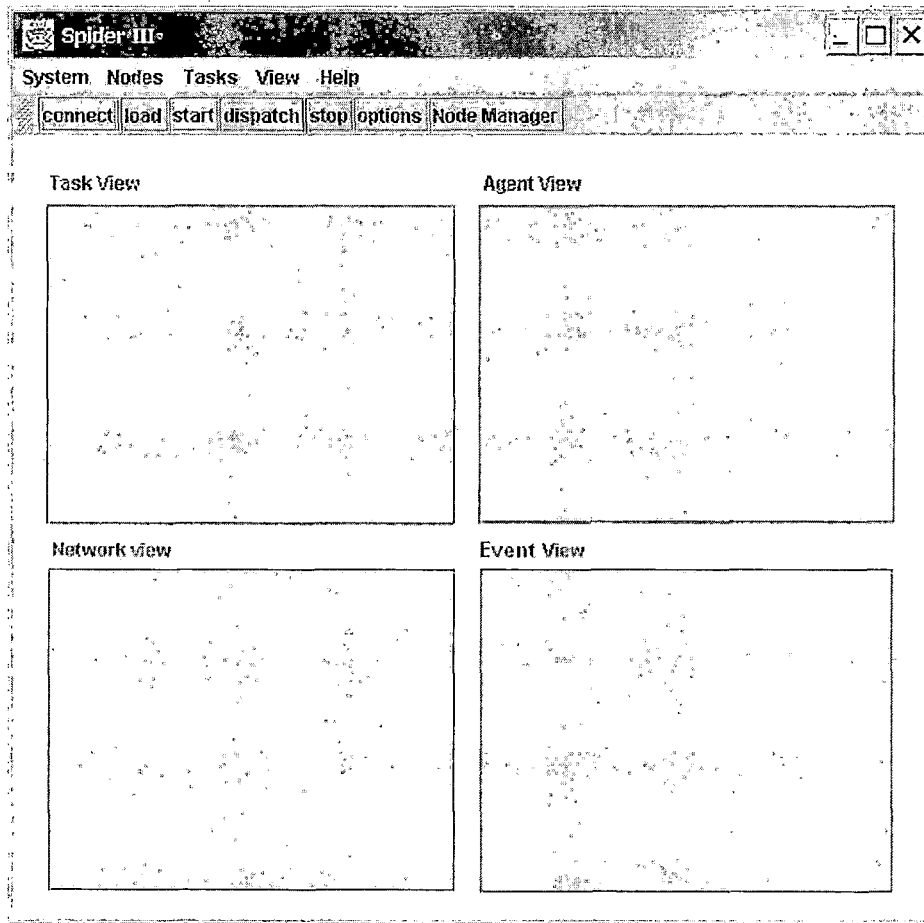


Figure 2.3. Spider III Graphical User Interface Main Frame

- Connect - when this menu is selected, the local computer tries to connect to other computers where Spider III system is installed and tries to dispatch the resource manager mobile agent to search for available computers. When the resource manager arrives other computers, it exchanges information with the resource manager on those computers.

- Disconnect - the node disconnects from the system, so that it will discontinue accepting any agents from other computers, including the one dispatched from it.
- Exit - close the application after disconnected from the system.

Tasks - contains submenu load, start, dispatch and stop.

- Load - selecting this menu item prompts the Load Task dialog. A Java class will be loaded into the system (see Figure 2.4).
- Start - initiates a task.

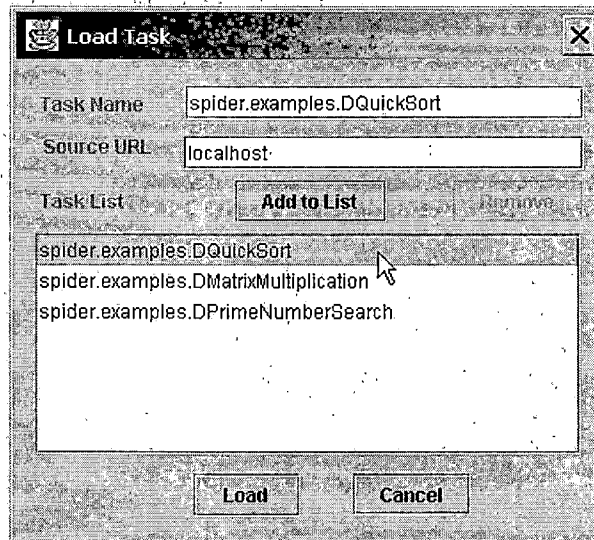


Figure 2.4. Load Task Dialog

- Dispatch - sends a task to be run on remote machine.
- Stop - aborts a running task.

View - contains submenu task, agent, network, and events.

- Task - displays the task view area.
- Agent - displays the system agents view area.
- Network - displays the network view area.
- Events - displays the contents of the event log file.

Help - contains submenu about, FAQs and contact.

- About - prompts the About dialog.
- FAQs - loads the FAQ page.
- Contact - prompts the Contact dialog.

2.3.1.3 Toolbar. The toolbar provides most functions of menus described above. It contains the following buttons: connect/disconnect, load, start, dispatch, stop, options, add, and remove. Clicking on a button has the same effect as selecting a menu item with the same text.

2.3.1.4 View Areas

Task View - provides information about each local task.

Agent View - shows name and status of all system agents running on the current node.

Network View - displays all nodes available to the current system and their utilizations.

Event View - provides log information of the system.

2.3.1.5 Dialogs And Frames. Dialogs are provided to view or control the behavior of the Spider III system.

Load Task - appears when the submenu "load" of the menu "tasks" are selected. It provides the user an opportunity to input the file name of the task, which should be a Java .class or .jar file.

About- provides version information about the software.

Contact - provides contact information about the author.

2.3.2 Functional Requirements

2.3.2.1 Add Nodes. A list of nodes can be provided in the configuration file, so that they can be automatically included into the system when the system starts up. Users can also input the address of a node to add it into the system after the system is started.

2.3.2.2 Connect. There should be at least one node available in its configuration file before a computer can

connect to the system. Otherwise the user is prompted with a dialog to ask for input.

2.3.2.3 Disconnect. The system should have been connected to the system before it is disconnected.

2.3.2.4 Load and Start/Dispatch a Task. A task that can be loaded into the system must be a Java class. It may exist as a .class or a .jar file. Furthermore, the class should have extended the Task class. Correct classpath is needed to locate a class file. URL is also needed if the class file resides on a remote machine.

2.3.3 Performance Requirements

The Spider III system supports multi-user and multi-task. Each user has her own terminal. Each user can submit multiple tasks simultaneously.

2.3.4 Software System Attributes

2.3.4.1 Reliability. The system should handle nodes failure and disconnection smoothly. The system should not crash frequently.

2.3.4.2 Security. The system's security mechanism is inherited from the underlying Agent framework. More advanced security is beyond the scope of this project.

2.3.4.3 Maintainability. All classes designed for the Spider III system shall be put in different packages from

those of the underlying agent framework. The Spider III system should avoid modifying the underlying agent framework as much as possible, and modifications should be documented in both the source code and a separate document clearly.

2.3.4.4 Portability. The Spider III system should be platform-independent. It should be able to be run in any system that has Java 1.2 or greater installed.

CHAPTER THREE

DESIGN OF THE SPIDER

III SYSTEM

The Spider architecture uses mobile agents as the basic components for the transparent distribution of computations, and uses static and mobile agents to manage resource sharing, load balancing, and fault tolerance. The Spider system provides a foundation for users to develop distributed applications through high-level API and services without knowing the underlying system architecture and protocols.

3.1 System Design

3.1.1 Core Agents in the Spider System

The Spider system consists of server nodes, worker nodes, and a Finder. A worker node is a place that provides an environment for executing local as well as migrated tasks. A server is a node where all worker nodes in that subnet are registered. Any node, generally the first started node, can be the server node. The roles of server node and worker node are exchangeable, which means that a server node could become a worker node if it is heavily loaded, and a worker node can become a server node after

the server node crashes, this will be discussed in later sections. On the other hand, all server nodes are required to register in a Finder, so that servers can communicate with each other across WANs. A task submitted from any location can take advantage of computational power in other domain of networks. The Finder is simply an RMI object registered at any host which all servers know how to communicate with.

Figure 3.1 presents the standard Spider host components to be installed on both worker nodes and server nodes in the network. It consists of three layers: agent execution environment, system agents, and user agents. The agent execution environment is made up of Java runtime environment and an agent framework, which provides the mechanisms of loading, dispatching, retracting, and executing agents on behalf of the user and supports communication between different agents. The system agent layer consists of a number of agents that are responsible for managing the system, including Synchronizer, UsageMonitor, Registry, Multicaster and SystemVerifier. Note that only Synchronizer and UsageMonitor agents in this layer are loaded at the system startup time. Other components are started or reactivated as needed dependent

over its task. This mechanism will be discussed in later section. The Synchronizer agent is also running on both worker node and server node for the purpose of fault tolerance. For a worker node it detects whether a server is available and for a server node, to detect whether there are multiple servers that co-exist in a local area network. Server components include Multicaster, Registry, and SystemVerifier. The Registry is located on a server to store addresses and usages of worker nodes and waits to provide service to worker nodes that request for task distribution. The Registry also communicates with the Finder so that changes in the local subnet can be updated at the Finder and that tasks initiated in a subnet can request resources from the WANs. The Multicaster and SystemVerifier work together with the Synchronizer to ensure that system failures can be detected and recovered.

3.1.2 The Task Distribution Protocol

Both worker nodes and server nodes provide the environment for executing locally as well as visiting agents, so a task can be initiated from either place. Figure 3.2 illustrates an example on how an application is distributed to the Spider system.

1. User initiates a master task in a node through the Task Viewer. This task is executed locally without consulting the server;
2. The master task sends a request to the Registry agent for available worker nodes by remote inter-agent communication or dispatching a messenger agent to the server;
3. The Registry replies with a list of addresses of available worker nodes;
- 3'. If there are not enough workers, the Registry talks with the Finder and gets the location of other servers on the WAN. A list of addresses of available servers is also appended to the message to be sent back to the master task;
4. The master task spawns a subtask for each worker node;
- 4'. If the Registry also replies with a list of server addresses, the master task talks with Registry agents on those servers to get more worker nodes
5. Subtasks are migrated to worker nodes;
6. A TaskID is returned to the master task for each spawned subtask. TaskIDs can be broadcasted to subtasks so that each subtask knows about each other.

Each subtask runs on its own host and can communicate with each other given that the TaskID is known.

- Each subtask finishes its execution, sends the results back to the master task, and then dies. The master task collects the results from all subtasks and then dies.

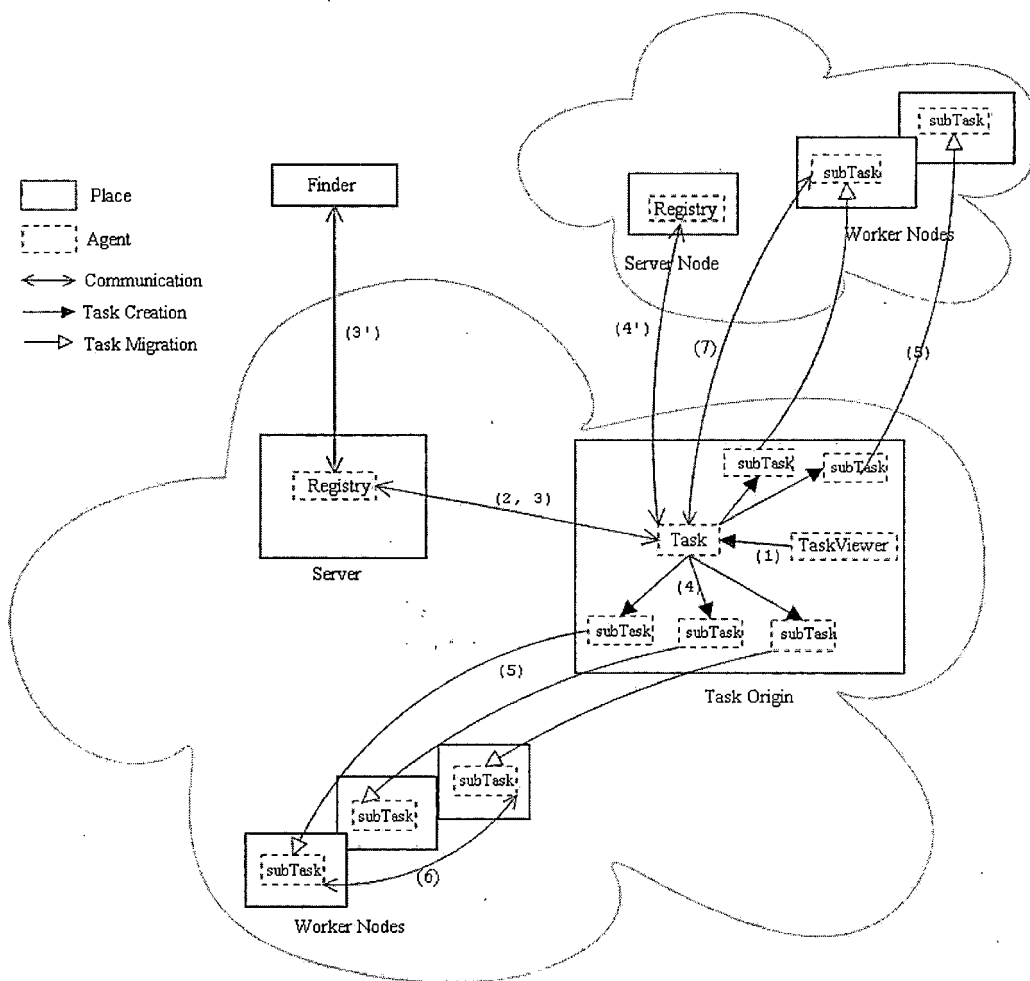


Figure 3.2. An Example of How an Application is Distributed to the Spider System

3.1.3 Load Balancing

Load balancing in each subnet is achieved by adapting the cyclic allocation mechanism used by PVM [12], with the addition of considering CPU utilization level as well as number of tasks. The Registry maintains the number of tasks running on each worker node and its CPU idle time. When a master task requests for spawning subtasks, it sends a message to the registry and gets a reply of a list of available worker nodes. A worker node will be considered available only when its CPU idle time satisfies a threshold value. Available worker nodes are further ordered according to the number of tasks running on that node. For nodes with the same number of tasks, are ordered by CPU idle time. Servers are informed of each other across WANs from the Finder.

The load balancing of the worker nodes in different WANs, however, is more difficult to achieve because there is no global order of all worker nodes. One possible solution is to register the loading level of each subnet in the Finder. For example, the average number of tasks running on each worker node could be used. When the Registry receives a request and the work loads in the local subnet is heavier (i.e., average number of tasks is larger)

than that of some other subnets, then the request will be forwarded to a Registry where the workload is lighter.

The Spider system does not provide dynamic load balancing. A task migrated to a worker node cannot be migrated again. This is due to the fact that the Java Virtual Machine on which most agent frameworks run does not support capturing the state of a thread, which would be a prerequisite for capturing and transferring execution state.

3.1.4 Fault Tolerance

System failure includes nodes failure and network failure. Nodes failure can be detected but is not distinguishable from network failure. We assume that nodes only suffer from crashes, which is a recovery failure. This type of failure causes the node to halt and loses its internal volatile state [16]. Also assume that networks are fully connected, i.e., any node can talk to each other directly. Detection and recovery of server failure is achieved with the cooperation of the Multicaster and the Synchronizer. The Multicaster multicasts the address of the server using multicast sockets. The Synchronizer checks multicast message to detect whether one or more servers are available. If a worker node does not receive multicast

message for a certain number of times, it considers that the server node has crashed and will promote itself to be a server node by starting appropriate agents. If two or more worker nodes promote themselves at the same time, they will detect each other and will demote themselves to worker nodes immediately. Each worker node waits a random amount of time before promoting itself, making it less likely for two worker nodes to promote themselves simultaneously. Once a new server is selected, all worker nodes will register in the Registry. Note that the information stored in Registry is only the address and usage of each worker node so it can be fully restored after a new server is selected.

Furthermore, the SystemVerifier on the server periodically checks that all registered nodes as well as the Finder are actually alive. A worker node is removed from the Registry after being unreachable for a certain number of times. Note that a faulty worker node can also be detected when a task attempts to migrate to that node. In that case, the Registry will also need to be notified, thus reducing the frequency of running the SystemVerifier, which is costly.

Similar to the SystemVerifier, the Finder uses a verifier thread to check whether all registered servers are

working properly. Besides, a newly selected server node can notify the Finder that the old server has been replaced. A communication failure with a Registry is also a hint that the server has already crashed.

Because Java Virtual Machine does not allow capturing execution state, there is no way to completely restore a failed task, i.e., it is impossible to provide failure recovery for tasks. However, it is feasible and reasonable to let the task register a `TaskFailed` event listener. The master task will be notified when a subtask fails and the programmer can specify what actions to take upon failure for example, to abort all subtasks including the master task, or to restart the failed task.

3.2 Architecture

The software components are organized by packages according to functions. The UML diagram in Figure 3.3 shows the overview of packages and classes in the Spider III system. To ensure a unique name, core packages of the Spider system are prefixed by `edu.csusb.spider`. Directly under this package are classes `Task`, `TaskID`, `Message`, Interface `MessageListener`, as well as a set of Exception classes which all inherit the `SpiderException` class. Also,

this package includes six sub-packages: *misc*, *pattern*, *finder*, *launcher*, *group* and *system*. *Task*, *TaskID* and *Message* class provide user APIs to develop distributed applications and will be discussed in detail in Chapter Four. *System* is the most important package in the Spider system. It contains all the system agents that we have discussed in previous sections and is essential for distributed applications to be run efficiently and transparently. The *group* package contains classes that implemented the group functionalities of *Task*. The *launcher* package contains classes that are used to start up and control the agent server. The *Finder* package contains classes that compose the *Finder*. The *pattern* package provides pre-defined programming paradigms to assist application development. Currently the only available pattern is the master-slave pattern but more patterns could be developed and added to this package. Finally other classes used by the Spider system are organized in the *misc* package. We now discuss the architecture of each package in the following sections.

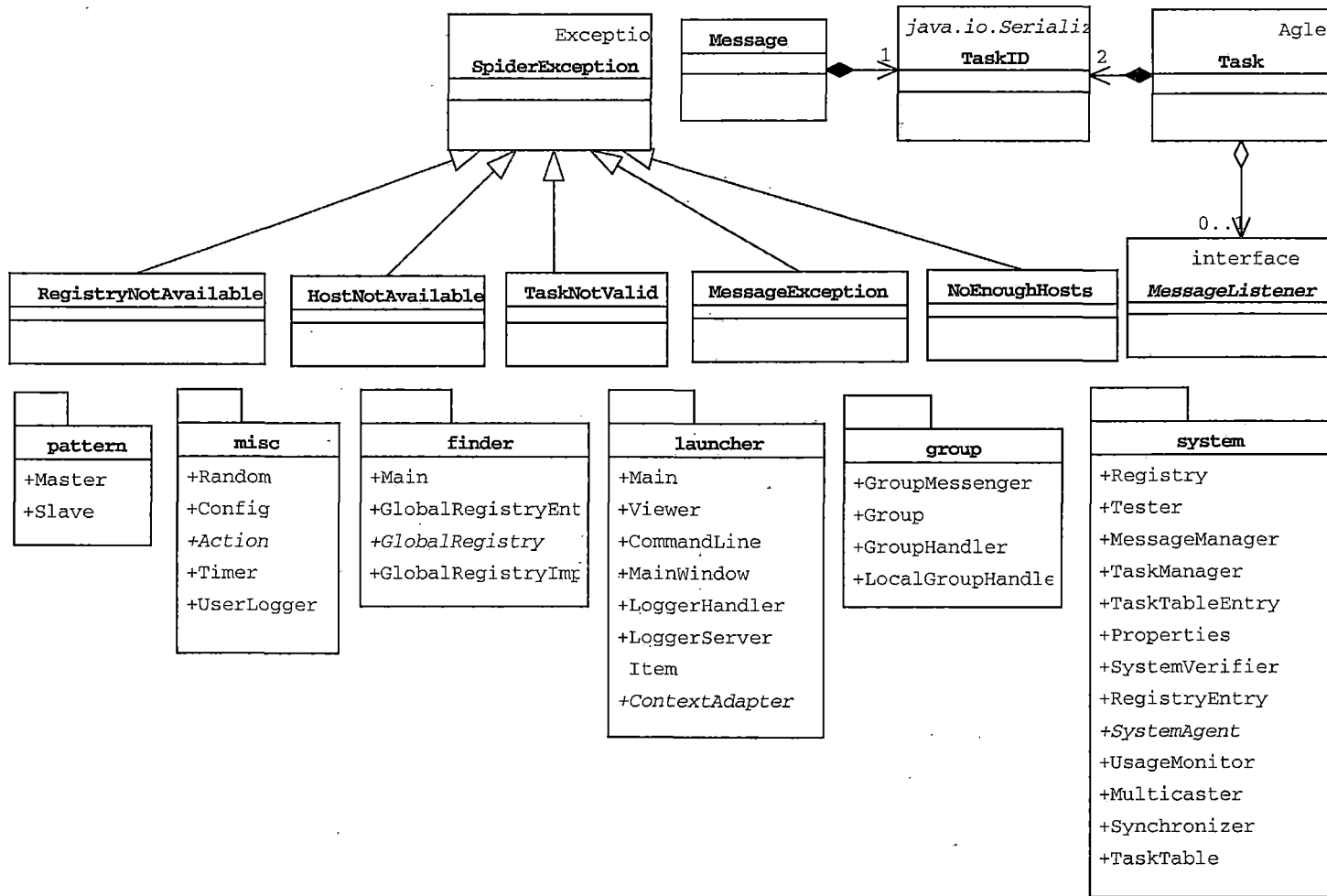


Figure 3.3. Overview of the Spider III Class Diagram

3.2.1 Package edu.csusb.spider.system

Figure 3.4 is a view of the *edu.csusb.spider.system* package. The *SystemAgent* is an abstract class that extends *Aglet*. It serves as the common parent class of all the other system agents that are used in the Spider, and does not possess any attributes and methods. The class *Registry*, *Multicaster*, *SystemVerifier*, *UsageMonitor* and *Synchronizer* represents the *SystemAgent* discussed before. Two other *SystemAgents* are not discussed in section 3.1: *Tester* and *TaskManager*. The *Tester* is created by *SystemVerifier* to test whether a worker node is still alive. The *TaskManager* is created when the first task is spawned and is destroyed when all tasks finish their jobs. The *TaskManager* is designed to provide services to local tasks. With a *TaskManager* running locally, the packet size to be transferred over network is reduced when migrating a user-defined task. The *TaskManager* provides services to *Tasks* through the *handleMessage()* method. Furthermore, the *TaskManager* caches the list of available hosts. When a task requests to spawn subtasks, the *TaskManager* first checks whether a local copy of host URLs are available and valid. If yes, then subtasks are spawned immediately, otherwise

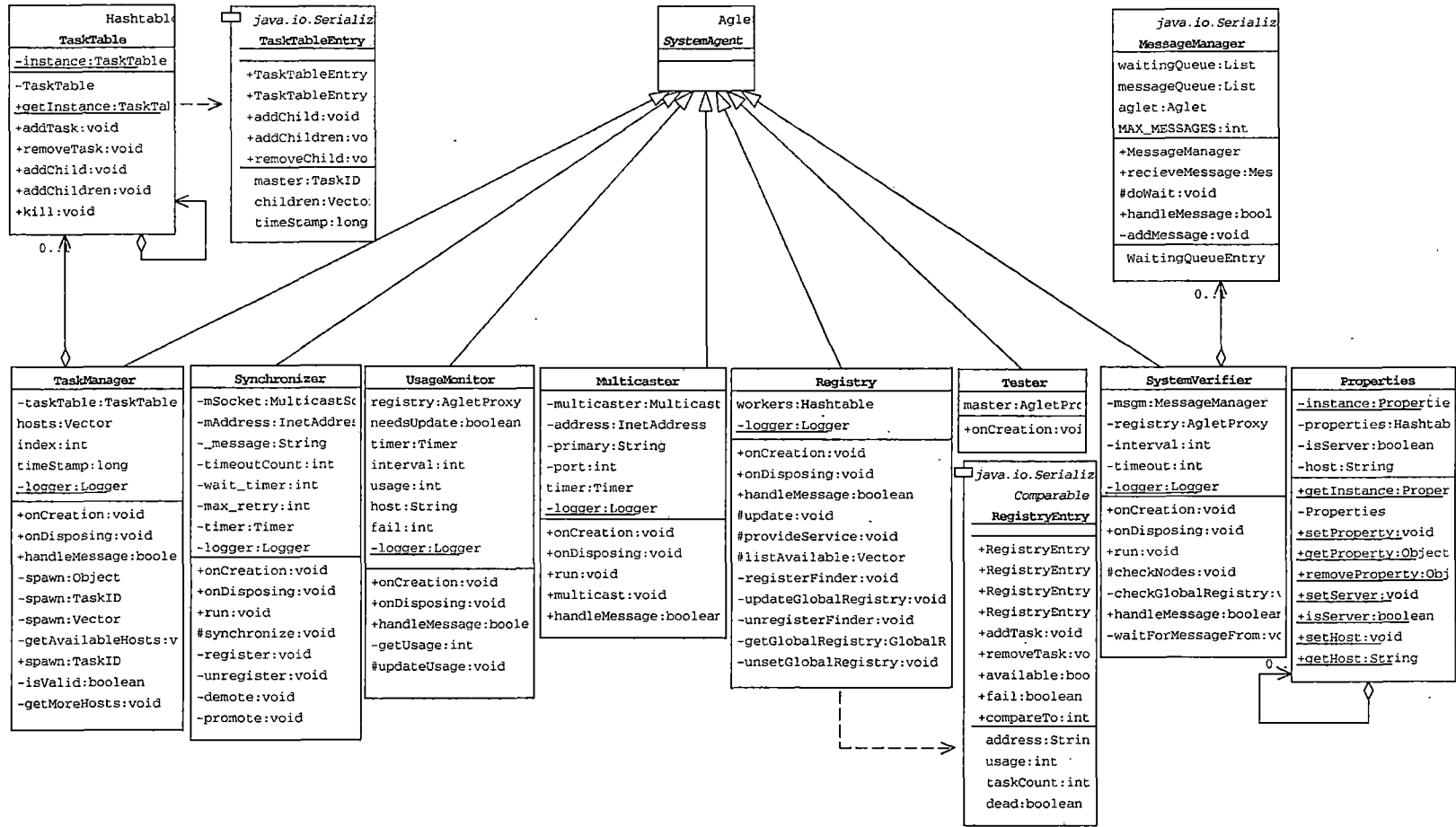


Figure 3.4. Class Diagram of the Package *edu.csusb.spider.system*

the TaskManager sends a message to the Registry to request for a list of hosts to spawn subtasks. A TaskTable is a singleton hash table, whose key is TaskID and value is TaskTableEntry. A TaskTableEntry stores parent TaskID and child TaskIDs of a task. The MessageManager manages the message queue and waiting queue of a task. The RegistryEntry stores *URL*, *usage* and *taskCount* of a host and composes the value of the hash table *workers* in the Registry. *Properties* is a singleton class that stores variables that can be used by the system across multiple agents.

3.2.2 Package edu.csusb.spider.finder

This package contains classes that compose the Finder. The Finder is implemented in RMI. The UML diagram is presented in Figure 3.5. GlobalRegistry defines interface of the remote object and GlobalRegistryImpl is the implementation of this interface. The GlobalRegistryImpl contains a hash table whose key is host URL and value is GlobalRegistryEntry. The Main registers an instance of GlobalRegistryImpl to the RMI registry and starts a ServerVerifier thread to check the validity of registered servers.

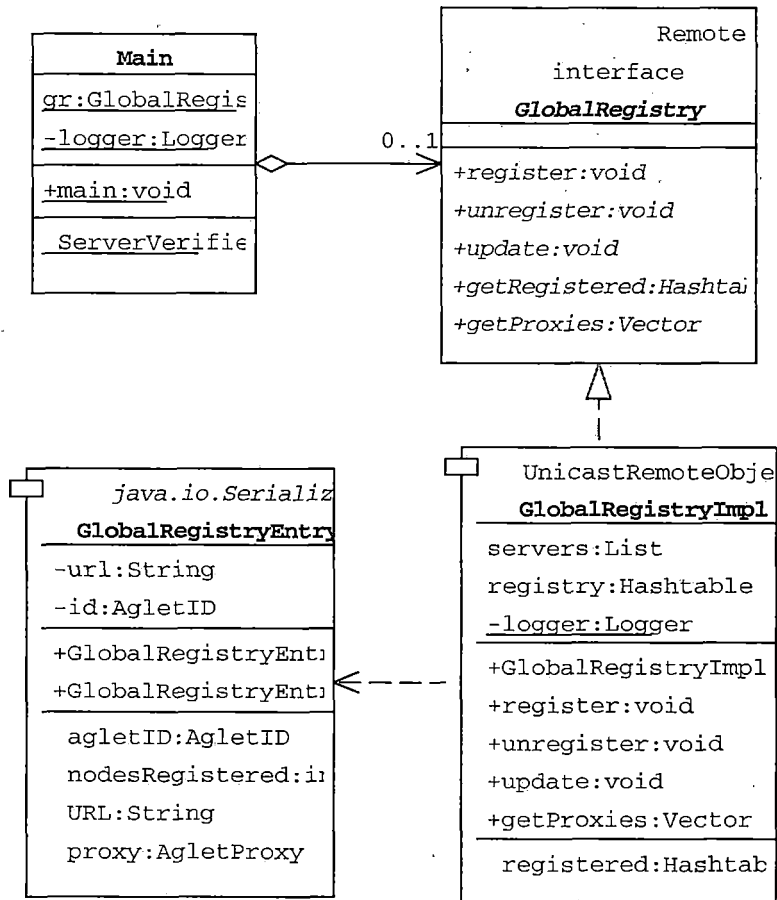


Figure 3.5. Class Diagram of the Package *edu.csusb.spider.finder*

3.2.3 Package *edu.csusb.spider.launcher*

The package *edu.csusb.spider.launcher* provides the mechanism of starting and controlling the Spider system. The Main class sets up system properties and bootstraps the Aglet server, on which a ContextListener is added to listen to context event. The ContextAdapter is an abstract class that implements most methods that are defined by the

ContextListener interface. The *added()* and *removed()* are two abstract methods that needs to be overridden when a class extends the ContextAdapter. Futhermore, if a LoggingServer is also initiated, subclasses of ContextAdapter can override the *logEvent()* method when other means of logging is desired other than command line display. CommandLine is a simple command line interpreter for the Spider system. It starts a thread to accept user input and listen to aglet context events. Viewer is a graphical user interface for the Spider system. A window is constructed using the MainWindow class, and user interactions are passed back to the Viewer class by calling the *Viewer#command()* methods. An Item extends TaskID, the only function implemented by Item is to provide a string representation of a task to be displayed on MainWindow. A LoggerServer can be started with a ContextAdapter to receive logging event. Basically the LoggerServer opens a ServerSocket and waits for connection from a client, which is generally a SocketAppender registered on a Logger. Once a connection is accepted, a new LoggerHandler thread is created, which in turns calls the *logEvent* methods of the *ContextAdapter* object. The UML diagram of this package is in Figure 3.6.

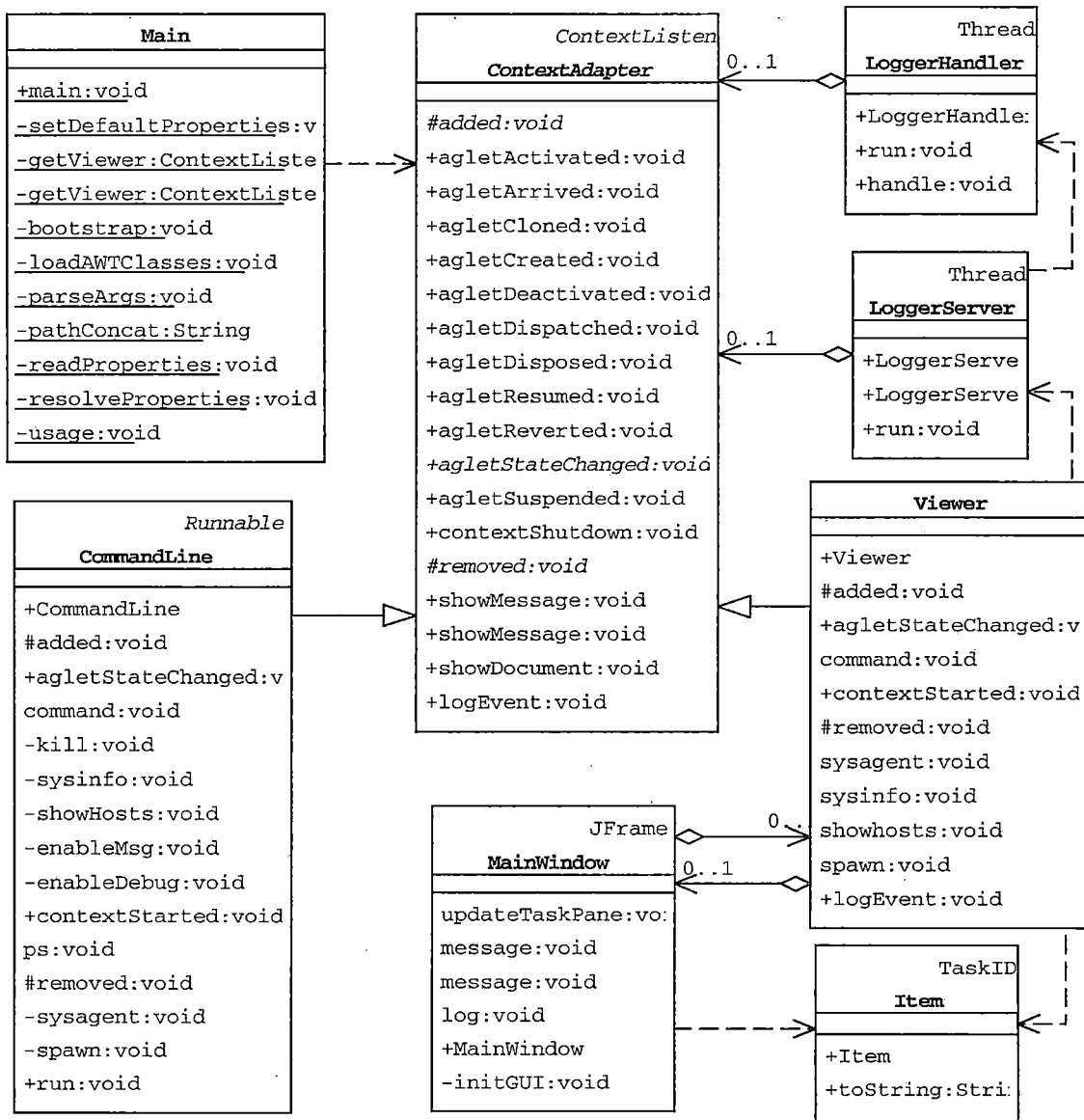


Figure 3.6. Class Diagram of the Package *edu.csusb.spider.launcher*

3.2.4 Package *edu.csusb.spider.misc*

This package provides utility classes for the Spider system. Config parses an XML document and sets up system properties. It is a singleton object so that it is

generally only parsed once at the system startup time, unless it is desired to reset properties at runtime, in which case the *refresh()* method should be called. Random generates a random number inside a range. UserLogger is created for application developers so that user messages can be differentiated from system messages. Timer is a thread that helps system set an alarm for a certain action in the future or periodically. Use of the Timer class involves implementing the Action interface by providing the *act()* method. The UML diagram of this package is in Figure 3.7.

3.2.5 Package edu.csusb.spider.pattern

This package is an extension of the Task class. It provides useful programming paradigms that can be applied directly to many applications. In this stage only Master/Slave pattern are provided. Master class overrides the spawn method to spawn a group of slave tasks and provides a *multicast()* method to send message to all slaves. On the other hand, the Slave class overrides the run method and provides a *sendMasterMessage()* method to send message back to master.

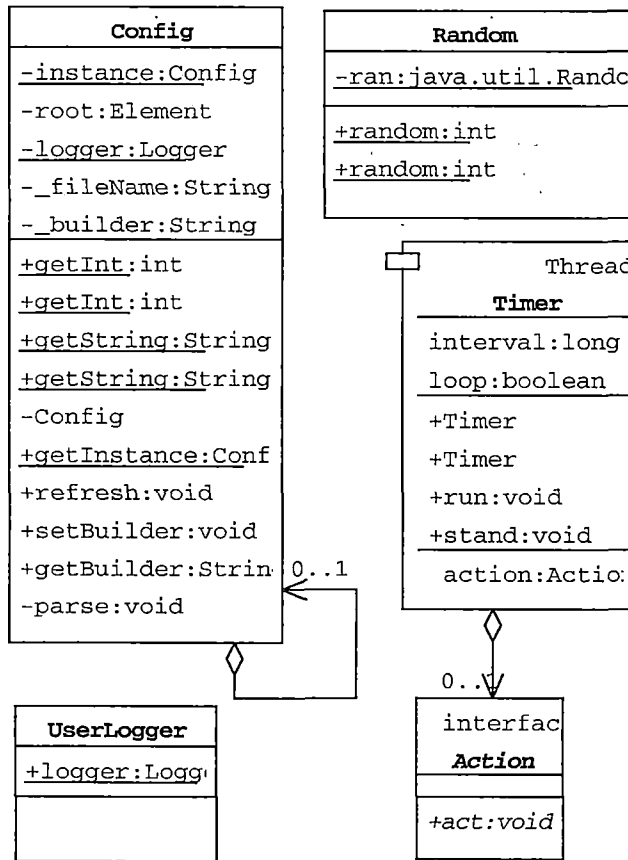


Figure 3.7. Class Diagram of the Package *edu.csusb.spider.misc*

3.2.6 Package *edu.csusb.spider.group*

This package helps to implement APIs for the group functions in the Task class. The LocalGroupHandler is used directly by a Task to perform group related functions. A LocalGroupHandler on behalf of a migrated task creates a GroupMessenger and dispatches it to the task origin when it firsts join the group. Then a handler to a remote GroupHandler is returned and stored in the

LocalGroupHandler. After that all the following group function calls will utilize the same handler. See Figure 3.8 for the class diagram of this package.

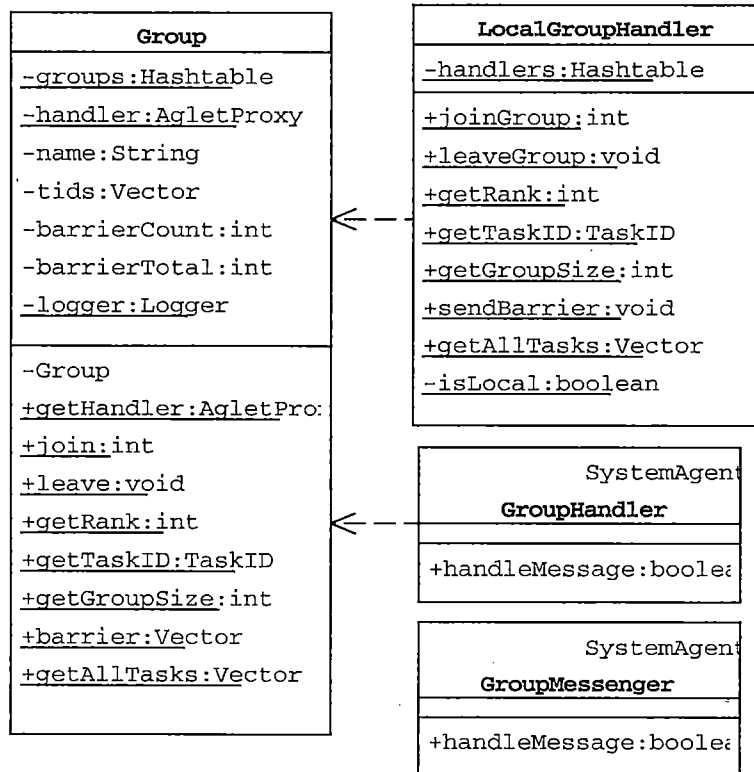


Figure 3.8. Class Diagram of the Package `edu.csusb.spider.group`

3.3 Detailed Design

This section contains the detailed design for the core components.

3.3.1 Synchronizer

Each node has a running synchronizer. Synchronizer starts other components as needed dependent on whether it is a server. The synchronizer on a node waits for multicast message from a server. If a message from a different server other than it knew is received, it either sends a register message to that server if it is a worker, or demotes itself to a worker if it is a server. If a worker does not receive a multicast message after trying for a certain number of times, then the worker node promotes itself to the server. Part of the pseudo code is shown in Figure 3.9.

```

public class Synchronizer extends SystemAgent {
    boolean isServer = false;
    String oldMessage = "";
    public void onCreation() { // initiation
        initialize multicast socket and properties
    }
    public void onDisposing() { // clean up
        unregister from the registry;
        close multicast socket;
    }
    public void run() {
        synchronize();
        set up a Timer to do synchronizer every 100 milliseconds;
        start Timer;
    }
    public synchronizer() {
        int timeoutCount = 0;
        wait for multicast message from server;
        if (timeout) {
            timeoutCount++;
            if (timeoutCount > 3)
                promote();
            return;
        }
        if (isServer and message != self) {
            demote();
        } else if (!isPrimary and message != oldMessage) {
            oldMessage = message;
            register();
        }
    }
    public void demote() {
        broadcast a message to all local system agents that the
server is going to demote
    }
    public void promote() {
        create system agents: Registry, SystemVerifer, and
Multicaster;
    }
    public void register() {
        extract server URL and registry AgletID from received
message;
        get AgletProxy from server URL and registry AgletID;
        store registry proxy into properties;
        send a start message to UsageMonitor;
    }
    public void unregister() {
        send a message to registry that the host is going to
leave;
    }
}

```

Figure 3.9. Pseudo Code of Synchronizer

3.3.2 Registry

The agent Registry stores information of all hosts in the same local area network and provides services by message passing. Part of the pseudo code is shown in Figure 3.10.

```
public class Registry extends SystemAgent {
    Hashtable workers;
    public void handleMessage(Message msg) {
        if (msg.sameKind("update"))
            update node status;
        else if (msg.sameKind("unregister"))
            workers.remove(msg.getArg("address"));
        else if (msg.sameKind("requestLAN"))
            replies with available LAN worker nodes ordered by
usage
        else if (msg.sameKind("request"))
            replies with available LAN worker nodes ordered by
usage and server nodes at other WANs
        else if (msg.sameKind("listAll"))
            replies with all nodes registered
        else if (msg.sameKind("lock"))
            label a node as locked
        else if (msg.sameKind("release"))
            release the lock of a node
        else if (msg.sameKind("fail"))
            label a node as unavailable;
            if a node is unavailable for more than 3 times,
remove from workers
        else if (msg.sameKind("registerFinder"))
            register server to the global registry
        else if (msg.sameKind("unregisterFinder"))
            unregister server from the global registry;
    }
}
```

Figure 3.10. Pseudo Code of Registry

3.3.3 Multicaster

The agent Multicaster declares the server's existence to its local network periodically. A server is selected in each LAN by the Synchronizer. These servers in different LANs consist of a server group. A server collects information of worker nodes in its local network, and updates the information to a global registry. The pseudo code is shown in Figure 3.11.

```
public class Multicaster extends SystemAgent {
    long interval = Config.getInt("multicaster.interval");
    public void onCreate(Object init) { // initialization
        initialize multicast socket and interval;
    }
    public void run() {
        pack host URL and registry AgletID into a datagram
        packet;
        multicast packet;
        set up a Timer to do multicast every interval
        millisecond;
        start Timer;
    }
}
```

Figure 3.11. Pseudo Code of Multicaster

3.3.4 UsageMonitor

This agent monitors local CPU usage and sends an update message to registry if the value has been changed ever since its last update. See Figure 3.12.

```

public class UsageMonitor extends SystemAgent {
    AgletProxy registry;
    boolean needsUpdate = true;
    int oldUsage = 0;
    Timer timer;
    long interval = Config.getInt("usageMonitor.interval");
    Public void onCreate(Object init) {
        Set up a timer to call updateUsage every interval
        milliseconds
    }
    public void handleMessage(Message mesg) {
        if (message kind is "start") {
            needsUpdate = true;
            updateUsage();
            start the timer;
        }
    }
    private updateUsage() {
        if (registry != null) {
            usage = local CPU idle time;
            if (oldUsage != usage || needsUpdate) {
                send a message to registry;
                oldUsage = usage;
                needsUpdate = false;
            }
        }
    }
}

```

Figure 3.12. Pseudo Code of UsageMonitor

3.3.5 SystemVerifier

The SystemVerifier is an agent that is responsible for checking whether a registered node is still alive. It periodically dispatches a tester agent to a registered node and waits for an acknowledge message. If an acknowledge message is not received after a given amount of time, that

node is labeled as unavailable in the registry. If a node is unavailable for a given number of attempts consecutively, it is removed from the Registry. At the same time, the SystemVerifier also validates that a Finder exists. The pseudo code is presented in Figure 3.13.

```
public class SystemVerifier extends SystemAgent {
    protected AgletProxy registry;
    long interval = Config.getInt("systemVerifer.interval");

    public void run() {
        while(Properties.isServer()) {
            Set hosts = registry.sendMessage(new Message("listAll"));
            CheckNodes(hosts);
            Check global registry;
            Try to deactivate for interval milliseconds;
            if fail then try to sleep for interval milliseconds
        }
    }
    checkNodes(hosts) {
        for all host in hosts {
            create a Tester agent;
            dispatch Tester to host;
            wait for acknowledge from Tester;
            if timeout then send a host fail message to the
registry;
        }
    }
    checkGlobalRegistry() {
        lookup RMI object GlobalRegistry;
        if succeed, send a registerFinder message to registry
    }
}
```

Figure 3.13. Pseudo Code Of SystemVerifier

CHAPTER FOUR

SPIDER USER APIS

The programming interface provided by the Spider system is intentionally similar to that of the widely used PVM system, but with syntax and semantics enhancement supported by Java and better matched to Java programming styles. The central interface through which most Spider applications interact with the system is the Task class, which is an agent itself. It provides functions to control the creation of additional tasks and to support communication among tasks. Users develop their own applications by extending the Task class. Users will need to provide the *doJob()* function which will be executed after a task is created. Users can also override the *onInit()* and the *onExit()* functions to define specific behaviors during the task initialization or termination period. The basic interface of Task is depicted in Listing 4.1.

```

public class Task {
    // methods to be overridden
    void onInit(Object arg);
    void onExit();
    void doJob();

    // identity
    public TaskID myTid();
    public TaskID parentTid();

    // send messages
    public void sendMessage(TaskID tid, Object mesg, String
tag);
    public void multicast(Vector tids, Object mesg, String
tag);

    // receive messages
    public Message recieveMessage(TaskID tid, String tag);
    public Message recieveMessage(TaskID tid, String tag,
long timeout);

    // create additional tasks
    public TaskID spawn(String className);
    public TaskID spawn(String className, Object arg);
    public Vector spawn(String className, int count);
    public Vector spawn(String className, int count, Vector
args);

    // group functions
    public int joinGroup(String groupName);
    public void leaveGroup(String groupName);
    public TaskID getTid(String groupName, int rank);
    public int getGroupSize(String groupName);
    public void broadcast(String groupName, Object mesg,
String tag);
    public void barrier(String groupName, int count);

    // asynchronous tasks
    public void addListener(Listener l);
}

```

Figure 4.1. Task Interface

4.1 Task Creation

The *spawn()* method provides the function to create additional tasks. It takes a string parameter indicating the name of a subtask class, which must also be a valid subclass of the generic Task class. Optionally the initial arguments and the number of tasks to be spawned can also be specified. The initial arguments are passed to the *onInit()* method, which users can override to define the behavior during task initialization. The underlying Spider system decides which worker node is chosen to spawn a child task. The ID of a newly spawned task is returned to the parent task upon successful creation. Alternatively, a number of same tasks can be started with a list of parameters, in which case a vector of IDs are returned.

4.2 Message Passing

Message passing in the Spider system is performed by calling the *sendMessage()* and *receiveMessage()* methods of the Task class. Unlike in PVM, there is no separate packing/unpacking operation needed for message passing. Any object that implements the *java.io.Serializable* interface can be used as the parameter to be sent to another task. To send messages, a caller only needs to provide a valid task

ID, without knowing where the task is physically running. An optional tag can be provided to label the message. The *multicast()* method broadcasts the message to all tasks specified in the vector except itself. Both the *sendMessage()* and the *multicast()* methods are asynchronous, which means the sending task proceeds immediately after the sending is initiated regardless whether it is received or not. However, if the destination ID does not represent a valid task, a *TaskNotValid* exception may be thrown. The *receiveMessage()* method with no timeout parameter performs blocking receive, while the *receiveMessage()* method with timeout parameter performs non-blocking receive. A positive number of timeout blocks the receiving task until the given number of milliseconds has past or the message arrives, whichever comes first. If the timeout value is zero, the method will immediately return no matter whether the message has arrived. Optionally a task ID and a tag can be provided to match with the coming message. The successful *receiveMessage()* method returns a *Message* object, whose contents can be extracted by calling the *getContent()* method of the *Message* class. The sender and the tag of the message can also be obtained by calling the *getSender()* and *getTag()* methods.

4.3 Group Functions

The Spider APIs support creation of dynamic task groups. A task can join or leave a group at any time. A named group is created at the first time when a *joinGroup()* method is called. *JoinGroup()* returns the rank of the task in the group. A task can join multiple groups. The method *getTaskID()* returns a task ID with a given group name and the rank of that task in the group. The function *getGroupSize()* returns the size of the group. The method *broadcast()* is similar to multicast but it uses the group name as parameter rather than task ID's. Calling *barrier()* method causes the task blocked until count members of the group has called the function. Similar to the barrier function of PVM, a count is required because with dynamic task groups the system cannot know how many members are in a group [17].

4.4 Asynchronous Tasks

To utilize the advantage of multi-thread programming of Java language, the Spider APIs support asynchronous tasks mechanisms, such as asynchronous message queues, allowing asynchronous processing of requests. For example, a *MessageListener* whose *onMessageArrival()* method being

overridden can be registered using the *addListener()* method so that a task can do its job while waiting for messages. Furthermore, *subTaskFailed()* method of *TaskStatusListener* allows the programmer to define the specific actions to take when a subtask fails and the *nodesAdded()* method of the *SystemEventListener* gives the programmer flexibility to write programs that can adapt to the dynamically changing system size. *TaskStatusListener* and *SystemEventListener* were not implemented in the current implementation.

4.5 Program Examples

An example of an application illustrating Spider APIs is shown in Listing 4.2. This application represents a simple Master/Slave program pattern. The Master is first started from the Task Viewer. It then spawns a set of slave tasks with initial arguments, each of which starts execution on a worker node in the system. The master task then multicasts a message to all slave tasks for processing. After that, the master task waits for all the slave tasks to return the result and combines them. On the other hand, the slave task is created by the master task with an initial argument. It then waits for the data from the master task. After processing, it sends the result back

to the master. All tasks then die after finishing execution.

```
/* file ExampleMaster.java */
package examples;
import edu.csusb.spider.*;

public class ExampleMaster extends Task {
    void doJob() {
        Vector args = initArgs();
        /* spawn N tasks */
        Vector tids = spawn("examples.slave", N, args);
        multicast(tids, data, "data");
        for (int i = 0; i < N; i++) {
            Message msg = recieveMessage(null, "result", -
1); //blocking recv
            processResult(msg.getContent());
        }
        exit();
    }
}

/* file ExampleSlave.java */
package examples;
import edu.csusb.spider.*;
public class ExampleSlave extends Task {
    Object init;
    void onInit(Object arg) {
        init = arg; // arg is passed from the master task
    }
    void doJob() {
        Message msg = receiveMessage(parentTid(), "data", -
1); // blocking receive
        result = processData(msg.getContent());
        sendMessage(parentTid(), result, "result");
        exit();
    }
}
}
```

Figure 4.2. Program Example

CHAPTER FIVE

SYSTEM TESTING AND PERFORMANCE ANALYSIS

In this section we evaluate the performance of the Spider system. We compared the performance of Spider with sequential program as well as PVM program when executing a matrix multiplication program. We also gave a mathematical analysis of the performance of the Spider system.

5.1 Testing Environments

All of the measurements were performed on 1 GHz AMD Athlon systems running Red Hat Linux 7.2 with 256 MB of memory, connected by a 10MB/sec Ethernet local network. JRE1.3.0.1 was used to run the Spider system and PVM3.4 was the system that we used for comparison.

5.2 Testing Methods

We implemented a distributed matrix multiplication algorithm described by Fox et al. [19] in Spider and compared to a sequential version of the program implemented in Java. We also run the matrix multiplication program provided in the PVM manual and compared its performance with sequential C program. We run the program with different problem sizes and record their execution time

separately. The problems range from 300×300 to 1200×1200 matrix sizes.

5.3 Testing Results

Figure 5.1 shows the sample screen of running the distributed matrix multiplication program.

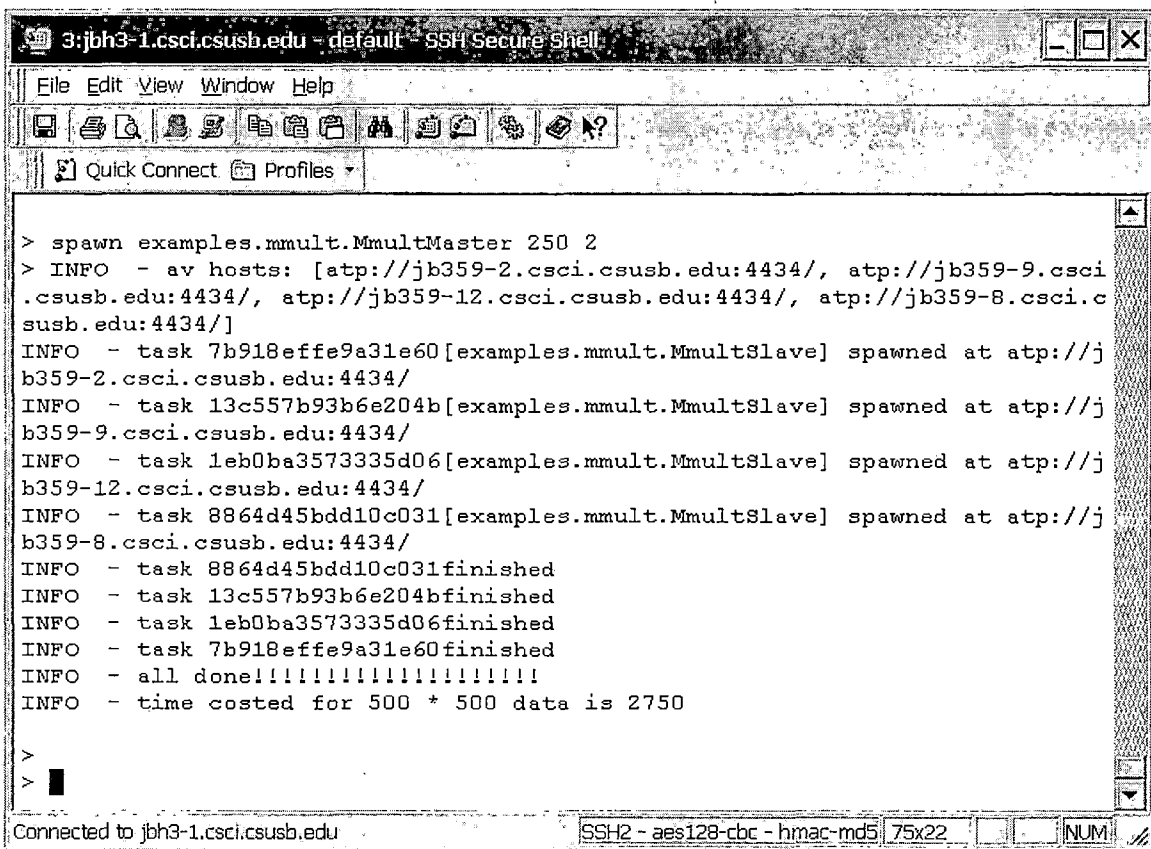


Figure 5.1. Sample Screen of Running the Distributed Matrix Multiplication Program

The results are shown in Table 5.1 and Figure 5.2, where speedup is defined to be the ratio of the execution time of the sequential algorithm to the execution time of

the distributed algorithm, efficiency is the percentage of the linear speedup and can be computed by the formula $\text{efficiency} = \text{speedup} / \text{number of nodes}$, and single task means sequential program. As we can see from Table 5.1 and Figure 5.2B, for small problem size (say 300×300), Spider system is slower than sequential program, while PVM system showed better performance than the sequential program. It is fairly reasonable when one considers that in Spider system a migrating task needs to transfer not only its data, but also its class definition. For medium problem size (say 500×500 to 700×700), an approximately equal speedup is achieved for both Spider and PVM system when 4 PCs are used. However, in the Spider system, the efficiency with large number of nodes is not as high as that with small number of nodes used (see Figure 5.2C), i.e., using 9 nodes has only a slightly better speedup than using 4 nodes. Again this is due to the heavy overhead to transfer multiple copies of Java bytecode to remote nodes. Finally, for large problem size (say 900×900 to 1200×1200), the distributed algorithm has the best speedup in the Spider system and the efficiency remains high with large number of nodes used. The maximum efficiency of PVM is about 0.7 in this experiment while efficiency of the Spider system is

0.9 (see Figure 5.2C). The last interesting fact that is worth pointing out is that matrix-multiplication programs implemented in Java are even faster than that implemented in C, which were observed in both sequential programs and distributed programs (see Figure 5.2A).

Table 5.1. Performance of Distributed Matrix Multiplication

Matrix Size	Tasks*	Spider			PVM		
		Time (sec)	Speedup	Efficiency	Time (sec)	Speedup	Efficiency
300 × 300	1	1.2	1.0	1.00	1.3	1.0	1.00
	4	1.5	0.8	0.20	0.6	2.2	0.55
	9	1.5	0.8	0.09	0.5	2.6	0.29
500 × 500	1	5.8	1.0	1.00	6.8	1.0	1.00
	4	2.8	2.1	0.52	2.4	2.8	0.70
	9	2.6	2.2	0.24	1.4	4.9	0.54
700 × 700	1	17.9	1.0	1.00	20.2	1.0	1.00
	4	6.9	2.6	0.65	7.5	2.7	0.68
	9	4.2	4.3	0.48	3.6	5.6	0.62
900 × 900	1	42.3	1.0	1.00	45.6	1.0	1.00
	4	12.5	3.3	0.82	17.4	2.6	0.65
	9	7.4	5.7	0.63	9.1	5.0	0.56
1200 × 1200	1	112.2	1.0	1.00	118.9	1.0	1.00
	4	29.6	3.8	0.95	41.8	2.8	0.70
	9	13.9	8.1	0.90	18.5	6.4	0.71

* Program with one task is in fact the sequential program, implemented in Java for Spider and in C for PVM.

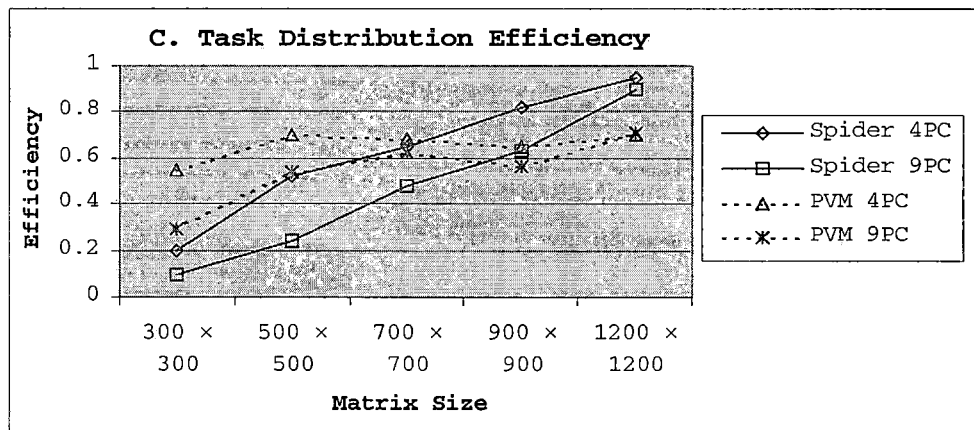
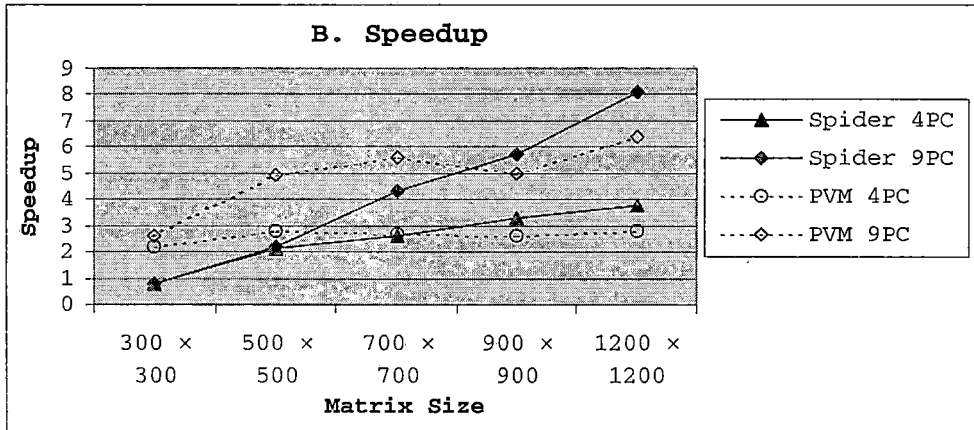
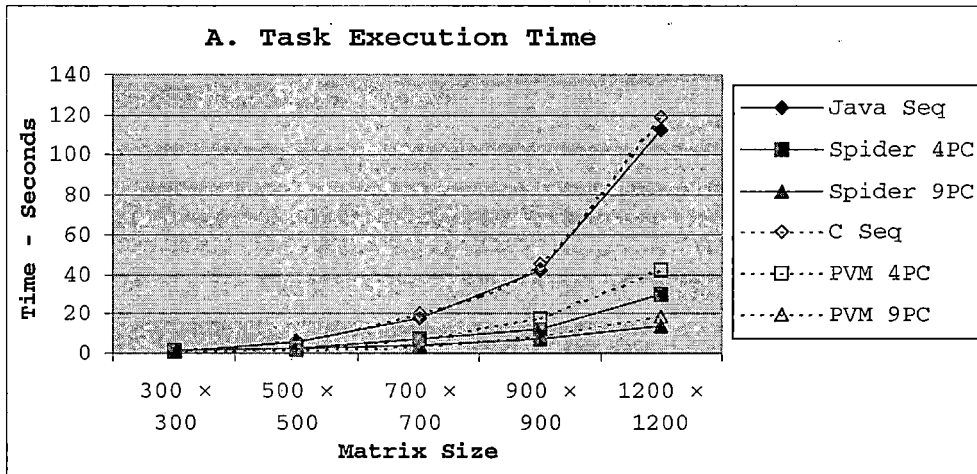


Figure 5.2. Comparisons Between Performance of Spider and PVM
 A. Comparison of execution time. B. Comparison of speedup. C. Comparison of distribution efficiency

5.4 Performance Analysis

In this section we perform the mathematical analysis of the task execution cost and system synchronization cost in terms of time, in order to discover bottleneck of the performance and to justify that system overhead is minimal in our design.

5.4.1 Execution Costs

1. Task execution, assume n subtasks are spawned and enough local resources are available
 - a. Initiate a task: the task manager sends a message to the server, selects the available hosts (negligible time), and returns.

$$t_{\text{init}} = 2t_{\text{msg}}$$

- b. Spawn subtasks: there are a total of n identical agents to migrate, but only one loading time is necessary because of the class cache

$$t_{\text{spawn}} = t_{\text{load}} + nt_{\text{mig}}$$

- c. Execute: assume each node takes the same length of time t_{exe} for execution
 - d. Finish: subagents send result back and combine results together

$$t_{\text{fin}} = n t_{\text{combine}}$$

The total time for a task with n subtasks to be finished is:

$$\begin{aligned}t_{\text{total}} &= t_{\text{init}} + t_{\text{spawn}} + t_{\text{exe}} + t_{\text{fin}} \\ &= t_{\text{load}} + 2t_{\text{msg}} + t_{\text{exe}} + n(t_{\text{mig}} + t_{\text{combine}})\end{aligned}$$

Where t_{load} is the time to initiate an agent, t_{mig} is the time to migrate an agent, t_{combine} is the time used to receive the result from a subtask and combine with the final result and t_{exe} is the time for one subtask to finish its execution.

2. If there is not enough resource available locally, the registry has to contact the Finder and then the worker nodes has to contact several other servers in the wide area network to gather computing resource. Assume the worker node has to contact a total of m servers to gather enough computing resource. In this case $2 * (m+1)$ messages are needed to be transmitted instead of two.

Therefore:

$$\begin{aligned}T_{\text{init}} &= 2(m+1)t_{\text{msg}} \\ t_{\text{total}} &= t_{\text{init}} + t_{\text{spawn}} + t_{\text{exe}} + t_{\text{fin}} \\ &= t_{\text{load}} + (2m+2)t_{\text{msg}} + t_{\text{exe}} + n(t_{\text{mig}} + t_{\text{combine}})\end{aligned}$$

To gain possible performance improvement, we can see that time spent on gathering resources and spawning new

tasks, as well as finishing the tasks, must be smaller than the time saved from execution on one computer. Simply assume that the execution time for a single-processor computer would be $n t_{exe}$, the following condition needs to be satisfied in order to gain performance improvement against sequential execution:

$$(n-1)t_{exe} > t_{load} + (2m+2)t_{msg} + n(t_{mig} + t_{combine})$$

Because t_{msg} is much less than t_{mig} , t_{mig} and $t_{combine}$, when n is large and m is small, which means most computing resource can be obtained from the local area network, approximately, we have:

$$t_{exe} > (1/n) * t_{load} + t_{mig} + t_{combine}$$

Under this assumption, the performance of our agent-based model is better than the non-distributed solution when the execution time of a subagent is greater than the sum of the average time to load an agent, the time to migrate an agent to a remote host, and the time to combine the results of two subagents.

5.4.2 Synchronization Costs

The total synchronization involves multicasting, dispatching tester agent, and computing CPU usage. Suppose that the frequencies for such action are λ , μ and ν

respectively. Also suppose that the probability for a worker node to update usage to server is ρ .

1. Synchronization cost for server

- a. Multicast its own existence to both LAN

$$t_{\text{declare}} = t_{\text{send}}$$

- b. Check aliveness of nodes in the local area network. This involves migrating the Tester agent and receiving the reply. Assume there are n nodes totally.

$$T_{\text{check}} = n * (t_{\text{migrate}} + t_{\text{receive}})$$

- c. Accept worker nodes update

$$T_{\text{update}} = n * \rho * t_{\text{recieve}}$$

2. Synchronization cost for worker node

- a. Detects presence of primary server

$$t_{\text{detect}} = t_{\text{recieve}}$$

- b. Compute its own CPU usage, and update the registry with a probability ρ :

$$t_{\text{update}} = t_{\text{compute}} + \rho * t_{\text{send}}$$

- c. Accepts the Tester agent and confirms that it is still alive.

$$t_{\text{confirm}} = t_{\text{accept}} + t_{\text{send}}$$

The total synchronization cost of the server is thus $\lambda t_{\text{send}} + \mu n (t_{\text{migrate}} + t_{\text{receive}}) + v \rho n t_{\text{receive}}$ and the synchronization cost of the worker is $\lambda t_{\text{receive}} + \mu (t_{\text{accept}} + t_{\text{send}}) + v(t_{\text{compute}} + \rho t_{\text{send}})$, where t_{receive} and t_{send} is the cost for sending and handling simple messages, t_{migrate} and t_{accept} is the cost for migrating and receiving a mobile agent, t_{compute} is the cost to compute local CPU usage. Understandably t_{migrate} and t_{accept} is much larger than t_{receive} and t_{send} . Luckily in our protocol the frequency for server to check nodes failure is rare, which means that μ value is small. Considering this fact, we can see that the synchronization cost for the server and worker are fairly balanced and is acceptable.

CHAPTER SIX

MAINTENANCE MANUAL

6.1 Obtaining a Copy

You can get a copy of this software from `ftp://spider.ias.csusb.edu/pub/spider_3.tar.z`, which contains all the necessary Java classes and configuration files. Source code and documentation are also included in this file. After you downloaded the file, you need to extract the tar file. Under a shell prompt, type:

```
tar xzf spider_3.tar.z
```

6.2 Directory Organization

Under the spider directory, there are six subdirectories: `bin`, `conf`, `docs`, `lib`, `public`, and `src`.

`bin`: This directory contains executable files for installation and server startup.

`conf`: This directory contains configuration files for the Spider system.

`docs`: This directory contains documentation for the Spider system.

`lib`: This directory contains all libraries required by the Spider system and user applications. "classes" subdirectory under `lib` contains classes developed for the

Spider system, namely the *edu.csusb.spider package*. Java runtime environment is not included and should be installed by the user separately.

`public`: This directory is where task class files that the user wants the Spider system to execute will be located.

`src`: This directory contains Spider source code.

6.3 System Requirements

Spider III has been tested on RedHat Linux6.x and Linx7.x and Windows 98. We believe it should take no extra efforts to be installed and run on other platform. The system requires Java 1.3 or up to be installed. At least 32 MB of memory is required (64MB recommended).

6.4 Installation

An ant script is used to help in compiling and installing the Spider system. Before running the script, you need to set up the `JDK_HOME` environment to the directory where your JDK is located, for example, `/share/java/jdk1.3.1`. Some system such as win95 may also require `ANT_HOME` environment variable to be setup. You can set it to the home directory of your Spider installation. After setting up the environment variable, change to the

bin directory and run the command: `ant install`. To recompile the source code, runs the command: `ant compile`. Or simply type "ant" to install after compile. Note that the installation will copy two files: `.java.policy` and `.keystore` to your home directory.

6.5 Configuration

If you just want to use default configurations, you can skip this section and go to section 6.6. However, in case you want to change the default settings, there are three configuration files under the `conf` directory. The `spider.xml` file is an XML document contains configuration of various Spider components. Explanation of each property is included within this file. The `aglets.props` file is for the Aglets system. Generally a Spider system user should not try to change this file. The `log4j.props` is the property file for log4j logger. An experienced user can figure out specific logging behavior, for example, to enable debug information of some components to be saved to a file or displayed on the console.

6.6 Running the Spider System

Change to the bin directory, run the command `spiderd` to start up a command line based Spider host and `xspider` to

startup a Java GUI based Spider host. Those commands can be run on any computer, and it will automatically detect whether a server is available in the same LAN and add itself into the virtual machine. If there is no server available, the Spider host will become a server. The Finder should be run on a computer whose address is specified in the spider.xml configuration file.

6.6.1 Startup Options

Type spiderd -help or xspider -help, you can see a list of startup options. Note that -nogui is the default option of spiderd.

-f <file.props>	Aglets properties file
-port <num>	port number (default 4434)
-verbose	verbose output
-nogui	omit AWT initialization
-daemon	run as a daemon
-help	print this message

6.6.2 Commands of Spiderd Console

After starting spiderd, wait for the server to startup, pressing "enter" to see the prompt ">", you can type "help" to see a list of commands acceptable by the spiderd console.

```
> help
```

help	Display this message.
shutdown	Shutdown the server.
reboot	Reboot the server.
msg on off	Message printing on/off.
debug on off	Debug message on/off.
spawn <task> [param list]	Spawn a new task.
clearcache	clear the memory cache of Task.
sysinfo	display system info.
hosts	display available hosts.
ps	List all tasks in the server.
kill <taskID>	Kill a task.
sysagent	Dispaly system agents.
>	

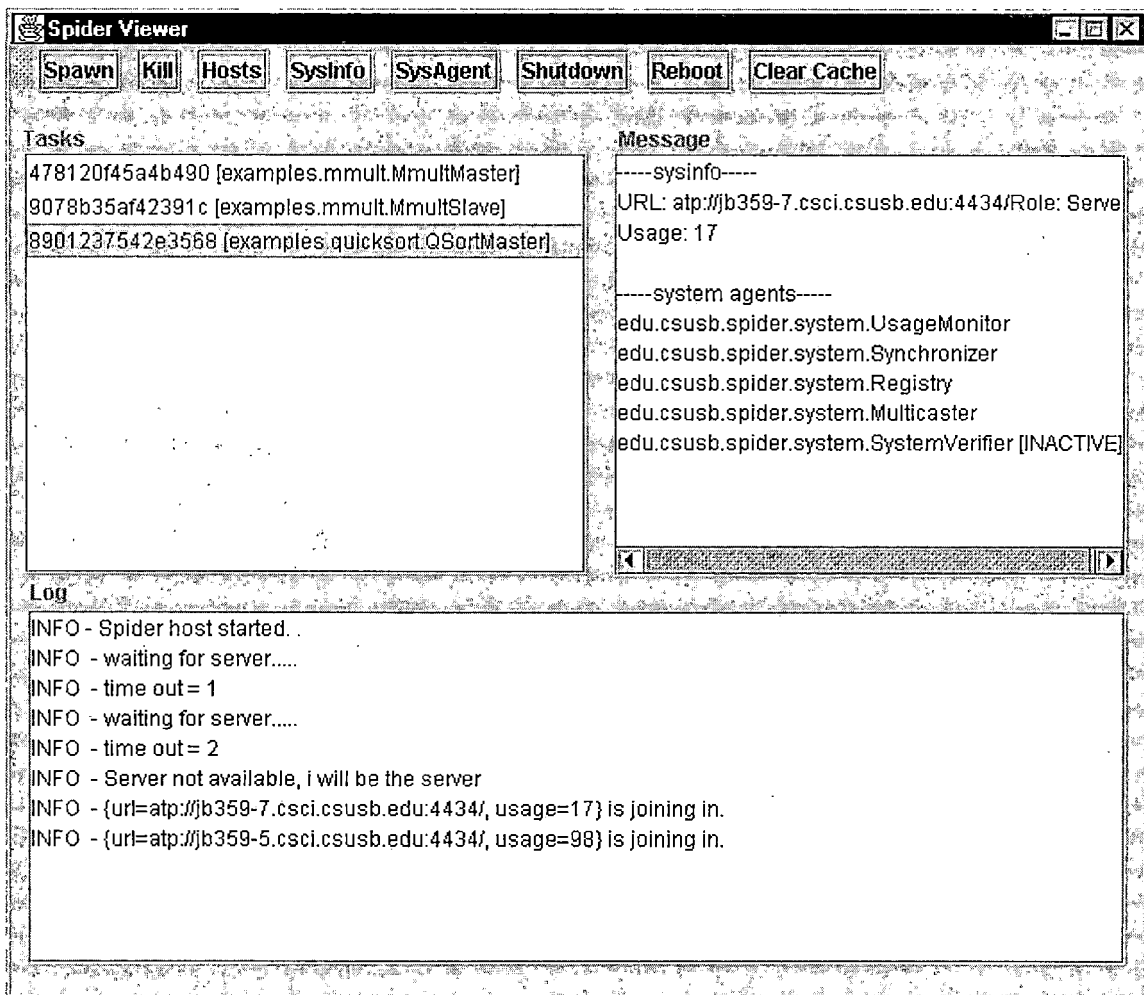


Figure 6.1. Xspider User Interface

6.6.3 Tools of Xspider Interface

The buttons Spawn, Kill, Hosts, SysInfo, SysAgent, Shutdown, Reboot, and Clear Cache has the same meaning as in the command line Spider console. There are three display areas in the Spider viewer window. The upper left panel displays the current running tasks, including their task ID and the class name. The upper right panel is the user

message area, where tracing statements of a user defined task is shown. When a user clicks on a button, such as SysInfo and SysAgent, the information will be displayed on this area. The lower display area is for system logging. User can choose to show different levels of information to be shown, for example, debug, info or error only, by editing the log4j.props under the conf directory.

6.7 Developing Applications on Spider III

6.7.1 Writing Source Program Using Spider APIs

The Spider system provides a set of APIs for programmers. we use a simple example to show how to develop an application using Spider APIs. The source code in Figure 6.2 demonstrates a master task creates several slave tasks and communicates with them. The HelloWorld class extends the generic Task class, and overrides the *onInit()* and the *doJob()* methods. The *onInit()* method tests whether the task is a parent task. If yes, then it parses command line arguments to initiate n. In the *doJob()* method, if it is a parent task, it first spawns n subtasks (line 29) and then starts a loop to receive messages from those subtasks (line 31) and prints out; if it is a child task, it sends its


```

1  package examples;
2
3  import edu.csusb.spider.Task;
4  import edu.csusb.spider.TaskID;
5  import edu.csusb.spider.Message;
6
7  public class HelloWorld extends Task {
8      private int n = 1;
9
10     public void onInit(Object init) {
11         if (getMaster() == null) // I am the parent
12             if (init != null) { // get the command line arguments
13                 String[] args = (String[]) init;
14                 n = Integer.parseInt(args[0]);
15             }
16     }
17
18     public void doJob() {
19         if (getMaster() == null) { // I am the parent
20             parentWork();
21         } else { // I am the child
22             childWork();
23         }
24         exit();
25     }
26
27     void parentWork() throws Exception {
28         try {
29             spawn("examples.HelloWorld", n); // create n children
30             for (int i = 0; i < n; i++) {
31                 Message mesg = receiveMessage(null, null, -1);
32                 String[] info = (String[]) mesg.getContent();
33                 System.out.println("Hello World from " + info[0]);
34                 System.out.println("os.arch = " + info[1]);
35                 System.out.println("os.name = " + info[2]);
36             }
37         } catch (Exception e) {}
38     }
39
40     void childWork() {
41         String [] info = new String[3];
42         info[0] = this.getHost();
43         info[1] = System.getProperty("os.arch");
44         info[2] = System.getProperty("os.name");
45         try {
46             this.sendMessage(parent, info, null);
47         } catch (edu.csusb.spider.TaskNotValid e) { }
48     }
49 }

```

Figure 6.2. HelloWorld.java

host URL and its operating system specific information to the parent (line 46).

6.7.2 Compilation and Deployment

To compile a source program, the user should add the directory `$SPIDER_HOME/lib/classes` into his classpath, where `$SPIDER_HOME` is the directory under which Spider is installed. Compiled class files need to be put under the directory `$SPIDER_HOME/public`, or any other directory specified by the `aglets.public.root` property in the file `$SPIDER_HOME/conf/aglets.props`. Alternatively, the user can simply put his source programs under the directory `$SPIDER_HOME/src/examples` and use ant script to compile and deploy by typing in "ant examples".

6.7.3 Running the Application

After the source program is successfully compiled and deployed, start the Spider console. Type in the command "spawn examples.HelloWorld <n>" to start the HelloWorld program. Figure 6.3 is what the output should look like.

```
[jruan@jb359-8 jruan]$ spiderd
> INFO - Spider host started
INFO - waiting for server.....
INFO - server is atp://jb359-2.csci.csusb.edu:4434/

> spawn examples.HelloWorld 3
> INFO - av hosts: [atp://jb359-2.csci.csusb.edu:4434/, atp://jb359-9.csci
.csub.edu:4434/, atp://jb359-8.csci.csusb.edu:4434/]
INFO - spawn examples.HelloWorld at atp://jb359-2.csci.csusb.edu:4434/
INFO - spawn examples.HelloWorld at atp://jb359-9.csci.csusb.edu:4434/
INFO - spawn examples.HelloWorld at atp://jb359-8.csci.csusb.edu:4434/
Hello World from jb359-2.csci.csusb.edu
os.arch = i386
os.name = Linux
Hello World from jb359-8.csci.csusb.edu
os.arch = i386
os.name = Linux
Hello World from jb359-9.csci.csusb.edu
os.arch = i386
os.name = Linux

> █
```

Connected to jbh3-1.csci.csusb.edu SSH2 - aes128-cbc - hmac-md5 75x22 NUM

Figure 6.3. Output of HelloWorld.java

CHAPTER SEVEN

CONCLUSIONS AND FUTURE

DIRECTIONS

7.1 Conclusions

The project, Spider III, presents architecture and protocol of a multi-agent based Internet distributed computing system, which provides a convenient development and execution environment for transparent task distribution, load balancing, and fault tolerance. This project presents the design and implementation of a prototype using the IBM Aglets software development kit (ASDK 2.0). The prototype implemented all the core agents, the task distribution protocol and a set of application programming interfaces (APIs), including a simple Task/Slave pattern. A graphical user interface is also developed to provide better visual and operational convenience.

The System supports transparent task distribution by providing a set of APIs. Load balancing is achieved using the adapted cyclic allocation mechanism, with the addition of considering CPU utilization level as well as number of tasks. Scalability is guaranteed by using two-level

registry services: all worker nodes in a LAN register to the server node in that LAN, and all servers in the WANs register to the Finder. Server crash is tolerable because the server is dynamically selected from all participating nodes.

To validate the design and to test the system performance, a distributed matrix multiplication is programmed on Spider and its execution time and distributed efficiency were compared with PVM and sequential programs. The results showed that although for small matrix size the Spider is slower than sequential program and PVM, the Spider system has better speedup and higher efficiency than PVM for appropriately large matrix size. Furthermore, the maximum efficiency of Spider (0.9) is significantly larger than the maximum efficiency observed in PVM (0.7). The results also showed that the Spider system is able to utilize CPU resources across multiple LANs and heterogeneous operating systems and architectures.

We analyze the task execution cost and system synchronization cost in terms of time, in order to discover performance bottleneck. We found that when the execution time of a subtask is greater than $(1/n) * t_{load} + t_{mig} + t_{combine}$, where t_{load} is the time to load an agent, t_{mig} is the

time to migrate an agent to remote host and $t_{combine}$ is the time to combine results of two agents together, our agent-based model is better than sequential version. We also show that the synchronization cost of the server is $\lambda t_{send} + \mu n (t_{migrate} + t_{receive}) + v \cdot \rho n t_{receive}$ and the synchronization cost of the worker is $\lambda t_{receive} + \mu (t_{accept} + t_{send}) + v(t_{compute} + \rho t_{send})$, where $t_{receive}$ and t_{send} is the cost for sending and handling simple messages, $t_{migrate}$ and t_{accept} is the cost for migrating and receiving a mobile agent, $t_{compute}$ is the cost to compute local CPU usage. Generally $t_{migrate}$ and t_{accept} is much larger than the other terms. In our protocol, we tried to reduce synchronization cost by reducing the value of μ , the frequency for a server to initiate checking nodes failure. Considering this fact, we conclude that the synchronization cost for server node and worker node are fairly balanced and is acceptable.

In the implementation of the Spider III system, we used object-oriented approach in designing and applied agent-oriented programming paradigms. Each component is programmed into an agent to separate different concerns. User APIs were intentionally made similar to that of the widely used PVM system to reduce learning efforts.

Furthermore, some advanced techniques are used in the implementation to provide flexibility and adaptability. System configuration of Spider III is done through property files and XML document, thus many of the system properties can be changed at runtime. Log4j package is used for system logging to enable/disable different levels of debug information. Compilation and deployment are done through ant scripts. Concurrent Version System (CVS) is used throughout the development time for version controlling.

7.2 Future Directions

Being a software developed by one person, the Spider system still has much to be improved. Furthermore, several key problems need to be solved to make the software a practical tool for use. Below are possible directions that can be extended from the result of this project.

- More APIs such as asynchronous event handling and client side IO redirection can be implemented by later developers. The current project only implemented a minimum set of APIs. Asynchronous events handling provides programmer the ability to receive notification of events and take certain actions at the same time when executing a task. This could let the

programmer automatically deal with nodes failure or easily develop an application that can be adapted to dynamically changing system environment. Besides the master/slave programming pattern, other patterns of distributed programming paradigms can be developed as a part of the *edu.csusb.spider.pattern* package to facilitate distributed application development. Client-side IO redirection gives the programmer ability to display all message in one screen or file.

- Multi-hop task distribution are not explicitly supported in the current implementation, i.e., A task migrated to a destination node cannot be migrated again. Although it could be done by interweaving the Aglets APIs and Spider APIs, a future improvement should standardize the protocol and include multi-hop task distribution as part of the Spider APIs.
- The current system startup is not so practical since it involves actually logon to a remote computer to start the server process. In the future the server process should be made into a daemon process. Although the current program can be run as a background process, a management console has to be developed to

control the background process. One problem that arises due to the lack of a control console on the daemon process is that although the Spider system supports multi-task and multi-user, two users might start their own copies of the server process on the machine where they intend to start the master task. A future version should separate the control console with the server process so that two different users can actually utilize the only server process running as a background service on the same computer.

- Finally, as a distributed computing system involves a large number of workstations across multiple domains owned by multiple organizations, security is no doubt the biggest concern. Although in the current Spider system, a basic level security is inherited from the Aglets framework by limiting migrated tasks to access local file systems, security breach still cannot be ignored. More work should be focused on analyzing and solving the security problem in the multi-agent architecture.

APPENDIX A

GLOSSARY

Agent	A software routine that waits in the background and performs an action when a specified event occurs. For example, agents could transmit a summary file on the first day of the month or monitor incoming data and alert the user when a certain transaction has arrived.
Agent Server	A component of an agent framework that provides basic services to agents, including inter-agent communication, agent migration, agent execution, agent security, etc.
Aglet	An agent is called aglet in the Aglets Software Development Kit
Aglets	An autonomous and mobile Java agent framework developed by IBM.
ASDK	Aglets Software Development Kit
ATP	Agent Transfer Protocol
CSUSB	California State University, San Bernardino
GUI	Graphical User Interface
IEEE	Institute of Electrical and Electronics Engineers
LAN	Local Area Network

Mobile Agent A software module that moves from host to host in a network

Node One of a group of machines that comprise a distributed system.

RMI Remote Method Invocation

Spider A distributed virtual machine system developed in Department of Computer Science, CSUSB.

SRS Software Requirement Specification

System Agent An agent that is used by the Spider system to exert system functions, such as resource manager.

URL Universal Resource Locator

User Agent An agent that is capable of executing user tasks. Users can rewrite the algorithm to achieve their own purposes.

WAN Wide Area Network

BIBLIOGRAPHY

- [1] IEEE std 830-1998 IEEE Recommended Practice for Software Requirements Specification
- [2] H.Yuh, *Spider: An Overview of an Object-Oriented Distributed Computing System*. Master Thesis, Department of Computer Science, California State University, San Bernardino, 1997
- [3] Koping Wang, *Spider II: A Component-based Distributed Computing System*. Master Thesis, Department of Computer Science, California State University, San Bernardino, 1997
- [4] Colin G. Harrison, David M Chess, and Aaron Kershenbaum. *Mobile Agents: Are they a good idea?* Technical Report, IBM, March 1995. URL <http://www.research.ibm.com/massdist/mobag.ps>
- [5] Jim Farley. *Java Distributed Computing*, O'Reilly & Associates, Inc, Sebastopol, CA, 1998. (ISBN 1-56592-206-9)
- [6] Rajkumar Buyya. *High Performance Cluster Computing*, volume 2, Prentice Hall, Upper Saddle River, New Jersey, 1999. (ISBN 0-13-013785-5)
- [7] Aglets SDK Document, <http://aglets.sourceforge.net>

- [8] Joseph P. Bigus and Jennifer Bigus. *Constructing Intelligent Agent with Java*, Wiley Computer Publishing, 1998. (ISBN 0-471-19135-3)
- [9] Andres S. Tanenbaum. *Distributed Operating Systems*, Prentice Hall, 1996, ISBN (7-302-02411-1)
- [10] Markus Straber, Jochim Baumann and Markus Schwehm, *An Agent-Based Framework for the Transparent Distribution of Computations*, In: H. Arabnia (ed.), Proc. 1999 Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'99), Vol I, CSREA, 1999, pp. 376-382, also available in mole group homepage.
<http://mole.informatik.uni-stuttgart.de/>
- [11] L. Dikken, et al. *DynamicPVM: Dynamic load balancing on parallel systems*. In W. Gentsch and U. Harms, editors, High Performance Computing and Networking, pages 273--277, Munich, Germany, April 1994. Springer Verlag, LNCS 797.
- [12] Adam J. Ferrari. *JPVM: Network parallel computing in java*. Technical Report CS-97-29, Department of Computer Science, University of Virginia, December 1997, also available in the JPVM Webpage,
<http://www.cs.virginia.edu/~ajf2j/jpvm.html>

- [13] Casas, J., et.al. *Mpvm: A migration transparent version of pvm*. 1995, *Computing Systems* 8, 2 (Spring), 171-216
- [14] Emin G. Sirer, et al. *Design and implementation of a distributed virtual machine for networked computers*, 17th ACM Symposium on Operating System Principles (SOSP'99), published as *Operating Systems Review* 34(5): 202-216, Dec. 1999
- [15] G. A. Geist, J. A. Kohl, and P. M. Papadopoulos. *PVM and MPI: A comparison of features*. *Calculateurs Paralleles*, 8(2), 1996.
- [16] M. K. Aguilera, W. Chen and S. Toueg. *Failure Detection and Consensus in The Crash-recovery Model*, Technical Report TR98-1676, Cornell University, Computer Science Department.
- [17] G. A. Geist et. al. *PVM: Parallel Virtual Machine - A User's Guide and Tutorial for Networked Parallel Computing*. MIT Press. 1994
- [18] Martin Fowler. *UML distilled: a brief guide to the standard object modeling language*, 2nd ed. Addison Welsey Longman Inc., 1999. (ISBN 0-201-65783-X).
- [19] G. C. Fox, S. W. Otto, and A. J. G. Hey. *Matrix Algorithms on A Hypercube I: Matrix Multiplication*. *Parallel Computing*, 4:17-31, 1987.