2001

Spider II: A component-based distributed computing system

Koping Wang

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

Part of the Software Engineering Commons

Recommended Citation


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 II: A COMPONENT-BASED DISTRIBUTED COMPUTING SYSTEM

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

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

by
Koping Wang
March 2001
SPIDER II: A COMPONENT-BASED DISTRIBUTED COMPUTING SYSTEM

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

by
Koping Wang
March 2001

Approved by:

Dr. Arturo Concepcion,
Chair of Computer Science

Dr. Owen Murphy

Dr. Kay Zemodeh
ABSTRACT

Spider II system is the second version implementation of the Spider project. This system is the first distributed computation research project in the Department of Computer Science at CSUSB. Spider II is a distributed virtual machine running on top of the Unix or Linux operating system. Spider II features multi-tasking, load balancing and fault tolerance, which optimize the performance and stability of the system.

Spider II system consists of four major components: Task Manager, Registry Server, Object Service Broker, and Object-Servers. The Task Manager manages the Object-Server’s usage. The Registry Server detects available Object-Servers. Object Service Broker controls the data communication and distributing the computation data to Object-Servers. Object-Servers are the network computers actually doing the computation.

Three distributed applications are developed in order to test the performance of Spider II: Distributed Matrix-Multiplication, Distributed Prime Number Search, and Distributed Quick Sort. The test shows that for large amount of data and complex computation, Spider II running
on a group of low-end network computers can have better performance than a single high-end server.
ACKNOWLEDGMENTS

First, I would like to thank my project advisor, Dr. Arturo I. Concepcion for his strong support in the development of this project. His valuable guidance and suggestions contributed substantially to this project.

And thanks also to Dr. Murphy and Dr. Zemoudeh, my project committee members, giving their insightful comments and suggestions, which make this project better.

I like to acknowledge Associated Studies Incorporated (ASI) at CSUSB, who helped to support this project through an ASI grant.

Finally, a special note of appreciation to my family - in particular, my parents and my wife. My parents' encouragement and financial support made my education possible. Because of my wife's love, prayers, and understanding, I was able to put all of my time and energy to complete this project.
TABLE OF CONTENTS

ABSTRACT ........................................ iii
ACKNOWLEDGMENTS ................................. v
LIST OF TABLES .................................. x
LIST OF FIGURES ................................. xi

CHAPTER ONE: INTRODUCTION

1.1 Introduction .................................. 1
1.2 Distributed Computing Systems ............... 1
1.3 Organization of Project Report ............... 4

CHAPTER TWO: SOFTWARE REQUIREMENTS SPECIFICATION

2.1 Introduction .................................. 6
  2.1.1 Purpose ................................... 6
  2.1.2 Scope ..................................... 6
  2.1.3 Overview ................................. 7
2.2 Overall Description ........................... 7
  2.2.1 Overview of Spider I System (Previous System) .... 7
    2.2.1.1 Object Service Broker ............... 7
    2.2.1.2 Registry Server .................. 8
    2.2.1.3 Task Manager Server .......... 8
    2.2.1.4 Graphical User Interface .......... 9
    2.2.1.5 Communication Protocols .... 9
  2.2.2 Spider II Product Perspective (New System) ..... 11
2.2.2.1 System Interfaces .... 11
2.2.2.2 User Interfaces .... 12
2.2.2.3 Hardware Interfaces .... 12
2.2.2.4 Software Interfaces .... 12
2.2.2.5 Communication Interfaces .... 12
2.2.2.6 Operations .... 13
2.2.3 Product Functions .... 13
  2.2.3.1 Registry Server and Load Balancing .... 13
  2.2.3.2 The Mirror Registry Server .... 14
  2.2.3.3 Ease of Adding New Service .... 14
  2.2.3.4 Reduced Communication Protocols .... 15
2.2.4 User Characteristics .... 17
2.2.5 Constraints .... 18
2.3 Specific Requirements .... 18
  2.3.1 Registry Server .... 18
  2.3.2 Task Manager .... 20
  2.3.3 Load Balancing .... 22
  2.3.4 Distributed Computing Service .... 23
    2.3.4.1 Distributed Quick Sort .... 23
    2.3.4.2 Distributed Prime Number Search .... 24
    2.3.4.3 Distributed Matrix Multiplication .... 24

CHAPTER THREE: DESIGN OF SIPDER II

vii
3.1 Architecture of Spider II . . . . . . . . . . . . . . . . . . 25
  3.1.1 System Design . . . . . . . . . . . . . . . . . . . . . . . . 25
  3.1.2 Software Architecture of Spider II . . . . . . . . . . . . 26
  3.1.3 Component Architecture of Spider II . . . . . . . . . . . . 28

3.2 Detailed Design . . . . . . . . . . . . . . . . . . . . . . . . . 29
  3.2.1 Task Manager . . . . . . . . . . . . . . . . . . . . . . . . 29
  3.2.2 Registry Server . . . . . . . . . . . . . . . . . . . . . . . 30
  3.2.3 Object Service Broker (OSB) . . . . . . . . . . . . . . . . 31
  3.2.4 Distributed Computing Service Applications . . . . . . . . 32
    3.2.4.1 Distributed Quick Sort . . . . . . . . . . . . . . . . . 33
    3.2.4.2 Distributed Prime Number Search . . . . . . . . . . . . 35
    3.2.4.3 Distributed Matrix Multiplication . . . . . . . . . . . . 37

CHAPTER FOUR: SYSTEM VALIDATION

4.1 Unit Test . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
4.2 Load Balance Test . . . . . . . . . . . . . . . . . . . . . . . . . . 43
4.3 Fault Tolerance Test . . . . . . . . . . . . . . . . . . . . . . . . . 44
4.4 System Test . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
4.5 System Performance Test . . . . . . . . . . . . . . . . . . . . . . . 45
  4.5.1 Testing Environments . . . . . . . . . . . . . . . . . . . . . . . 45
  4.5.2 Testing Method . . . . . . . . . . . . . . . . . . . . . . . . . . 46
  4.5.3 Testing Results . . . . . . . . . . . . . . . . . . . . . . . . . . 46
4.5.3.1 Matrix Multiplication ........ 46
4.5.3.2 Prime Number Search ......... 48
4.5.3.3 Quick Sort .................. 50
4.5.4 Performance Analysis .......... 51

CHAPTER FIVE: MAINTENANCE MANUAL

5.1 System Requirement ............... 53
5.2 Installation and Configuration ...... 53
5.3 Running Spider II System .......... 56
5.4 Adding a New Application .......... 56
   5.4.1 Client Program ................ 57
   5.4.2 Server Program ................. 58

CHAPTER SIX: FUTURE DIRECTIONS AND CONCLUSIONS .... 59

APPENDIX A: GLOSSARY ............... 63
APPENDIX B: SOURCE CODE ............ 68
REFERENCES .......................... 133
LIST OF TABLES

Table 2.1. The Protocols Used in Spider .................. 10
Table 2.2. Priority Rules ................................. 20
Table 4.1. Unit Test ........................................ 40
Table 4.2. Load Balancing Test ............................. 43
Table 4.3. Fault Tolerance Test ............................ 44
Table 4.4. Maximum Data Size of Spider II Applications . 45
Table 4.5. Comparison of Matrix Multiplication ............ 46
Table 4.6. Comparison of Prime Number Search ............ 48
Table 4.7. Comparison of Quick Sort ....................... 50
Table 5.1. "profile.h" file ................................. 54
Table 5.2. Commands of Spider II .......................... 55
LIST OF FIGURES

Figure 2.1. Protocol of the Distributed Computation of Spider System ............ 10
Figure 2.2. System Interface of Spider II .......... 12
Figure 2.3. Object Engine Design ............... 15
Figure 2.4. The Protocols of the Distributed Computation in Spider II ............ 16
Figure 2.5. Use Case Diagram ................. 17
Figure 2.6. Registry Server Interface ............ 19
Figure 2.7. Task Server: Initial Value .......... 20
Figure 2.8. Task Server: Get Information from Registry 21
Figure 2.9. Task Server: After the Task ID is Assigned 21
Figure 3.1. System Design of Spider II ........... 26
Figure 3.2. Software Architecture of Spider II ........ 28
Figure 3.3. Component Architecture of Spider II ........ 29
Figure 3.4. Task Manager Class Diagram ........... 30
Figure 3.5. Registry Server ................. 31
Figure 3.6. Object Service Broker ............... 32
Figure 3.7. Application: Prime Number Search ........ 33
Figure 3.8. Algorithm of Distributed Quick Sort ........ 35
Figure 3.9. Algorithm of Distributed Prime Number Search ................. 37
Figure 3.10. Sub-Matrix Algorithm ............... 38
Figure 3.11. Distributed Matrix Multiplication Algorithm ................. 39
Figure 3.12. The Final Process in Client . . . . . . . . 39
Figure 4.1. Comparison of Matrix Multiplication . . . . 47
Figure 4.2. Comparison of Prime Number Search . . . . 49
Figure 4.3. Comparison of Quick Sort . . . . . . . . . . . . . 51
CHAPTER ONE: INTRODUCTION

1.1 Introduction

Spider project is the first distributed computation research project in the Department of Computer Science at CSUSB. It is initially developed by Han-Sheng Yuh in his Master’s thesis in 1997 [10]. It is an object-oriented distributed system, running C++ on UNIX or UNIX-like operating system. In his thesis, Han-Sheng introduced the overall goal for Spider and he defined major components for the system. He also implemented the communication protocols for the distributed computation on Spider. The system was tested on SGI Indigo workstations and Irix operating system.

The second version of Spider, Spider II, continues to improve existing features and additional various features of Spider, which include improving the communication protocol performance, load balancing, stabilizing the multi-tasking operation, and adding more functionalities. The details of Spider II will be discussed in Chapter Two Software Requirements Specification.

1.2 Distributed Computing Systems

A distributed computing system is a system that is a collection of computers, which run their own operating
system or distributed operating system without having a global memory or a single clock, and the computers communicate with each other by exchanging messages over a network [8]. In recent years, distributed computation on networks and workstations become more and more popular. This is because personal computers and network workstations have very good price and performance and with higher network bandwidths, they provide very good environment for developing distributed system. A distributed system can provide a powerful computation facility by using a network of low-end workstations.

There are several well known distributed system projects in existence as of today that performs various tasks and applications: Beowulf, MOSIX, MPI and PVM.

Beowulf project was started in 1994 at The Center of Excellence in Space Data and Information Sciences (CESDIS). CESDIS was operated for NASA. By definition, Beowulf is a system, which usually consists of one server node, and one or more client nodes connected together via Ethernet or some other network. It is a system built using commodity hardware components, like any PC capable of running Linux, standard Ethernet adapters, and switches [7]. The Beowulf Project is now hosted by Scyld Computing Corporation, which
was founded by members of the original Beowulf team with a mission to develop and support Beowulf systems in larger commercial arena [1].

MOSIX is a software package that was specifically designed to enhance the Linux kernel with cluster computing capabilities. The core of MOSIX features load-balancing, memory ushering, and file I/O optimization algorithms that respond to variations in the use of the cluster resources [6]. MOSIX is developed in the Institute of Computer Science of The Hebrew University of Jerusalem, Israel.

MPI stands for Message Passing Interface, which is a library specification for message-passing proposed as a standard by a broadly based committee of vendors, implementers, and users. The MPI standardization involved about 40 organizations mainly from the United States and Europe. The main advantages of establishing a message-passing standard are portability and ease-of-use. The definition of a message passing standard provides vendors with a clearly defined base set of routines that they can implement efficiently [5]. MPI is managed by Mathematics and Computer Science Division, Office of Computational and Technology Research of the U.S. Department of Energy.
PVM (Parallel Virtual Machine) is an integrated set of software tools and libraries that emulates a general-purpose, flexible, heterogeneous concurrent computing framework on interconnected computers of varied architecture [3]. The overall objective of the PVM system is to enable such a collection of computers to be used cooperatively for concurrent or parallel computation. The PVM project began in 1989 at Computer Science & Mathematics Division of Oak Ridge National Laboratory. The general goals of this project are to investigate issues in, and develop solutions for, heterogeneous concurrent computing.

1.3 Organization of Project Report

This project report is organized into six chapters. Chapter One is the introduction. Chapter Two is the Software Requirements Specification (SRS), which include the overview of Spider I and II, software functions, and the hardware and software requirement for Spider II. The SRS follows the IEEE recommended practice for SRS (IEEE Std 830-1993). Chapter Three contains the design of Spider II. We will discuss the detailed design of every component of the system in this chapter. Chapter Four is testing, which shows how we tested the system and measure the performance of the system. Chapter Five is the maintenance manual. It
will describe how to maintain the system and how to develop distributed programs using Spider II. Chapter Six is the conclusion and suggestions for future development directions.
CHAPTER TWO: SOFTWARE REQUIREMENTS SPECIFICATION

2.1 Introduction

2.1.1 Purpose

This Software Requirements Specification is for the Spider II Project, which is the second version of the implementation of the Spider Distributed System. Spider II is the Master Project for Koping Wang as the final requirement for the degree of M.S. in Computer Science.

2.1.2 Scope

Spider II is a distributed virtual machine running on top of the UNIX or UNIX-like (LINUX) operating system. It permits a collection of computers connected together by a network to be used as a single large parallel computer. Spider II features multi-tasking, load balancing and fault tolerance, which optimize the performance and stability of the system. For a complex computation, Spider II can have better performance than a single high-end server.

There are some limitations of Spider II. First, Spider II has been tested only within the CS network of CSUSB. Second, in this phase of the implementation, the platform for Spider is limited on UNIX or UNIX-like operating system.
2.1.3 Overview

This Software Requirement Specification contains the following: We describe first the system design and components of the previous implemented Spider system. Then we describe the Spider II product perspective, its hardware and software requirements, user interface, and communication protocols. Then we show what improvement and enhancements were performed on down to Spider I to develop Spider II.

2.2 Overall Description

2.2.1 Overview of Spider I System (Previous System)

In the previous Spider project, the distributed computation service (DCS) was implemented. The DCS was designed in an object-oriented approach and written in C++ and Java programming language. The DCS has the following components: Object Service Broker (OSB), Registry Server (RS), Task Manager (TM), and Graphical User Interface (GUI).

2.2.1.1 Object Service Broker

OSB plays an important role in the Spider system. The main functions of the OSB are:
(a) to locate the registered object-servers from the Registry Server according to the client's request;
(b) to submit the job to the TM for monitoring the execution.
(c) to activate the remote object-server and convey the client's request to that object-server.
(d) to notify TM that the job is finished and deactivate the object-server.

When an object-server needs to request for another object-server(s), it also requires the OSB to locate and activate the remote object-servers. OSB simply talks to TM requesting for available object-server, then activates the object-server on the remote machine.

2.2.1.2 Registry Server

The main function of the Registry Server is to manage an object-server database, and to provide a list of available registered object-servers. RS returns the hostname and corresponding port number to OSB's request.

2.2.1.3 Task Manager Server

In order to manage the available servers for each task, Task Manager needs to keep track of all tasks' activities. TM contains two servers. One is TM server and the other is mirror TM. The mirror TM keeps a copy of the
information of the TM. In case the TM goes down, the mirror TM will take over TM's job. OSB has two options for contacting the TM. First, OSB tries to contact the TM server, if TM is down or no response, OSB will try to contact the mirror TM. If both servers are down, then the operation is aborted because the object-servers cannot operate properly without the TM.

2.2.1.4 Graphical User Interface

To provide a friendly user interface, the GUI is implemented in Java programming language. Java applets are able to run on any platforms where there is a Web browser. Therefore, anyone is able to access the Spider’s DCS through the Internet, if they are authorized users.

2.2.1.5 Communication Protocols

RPC and BSD Socket interfaces are the communication mechanisms used for the Spider system. RPC provide the data conversions for clients and object-servers. The BSD Socket interface is used for the communication of object-servers. The TCP and UDP protocols are needed to suit different needs in the implementation. Table 2.1 shows the protocols used for communications within Spider.
Table 2.1. The Protocols Used in Spider

<table>
<thead>
<tr>
<th>Type</th>
<th>Protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client - server</td>
<td>RPC</td>
</tr>
<tr>
<td>OSB - Registry</td>
<td>UDP</td>
</tr>
<tr>
<td>OSB - Task Manager</td>
<td>TCP</td>
</tr>
<tr>
<td>Task - Task Manager</td>
<td>TCP</td>
</tr>
<tr>
<td>Task - task</td>
<td>TCP</td>
</tr>
<tr>
<td>SVR_OSB - Task Manger</td>
<td>TCP</td>
</tr>
<tr>
<td>task - Java Daemon</td>
<td>UDP</td>
</tr>
</tbody>
</table>

To understand the communication protocol of the distributed computation in the Spider system, Figure 2.1 illustrates the steps of the distributed computation.

Figure 2.1. Protocol of the Distributed Computation of Spider System

When a client requests a service, it sends the request to OSB, then OSB asks the Registry Server for the registered
server. After OSB gets the reply from RS, it sends a notification to TM and gets the Task ID from TM. At the same time, the mirror TM keeps a copy of the information in TM. Then OSB starts to send the job to Object-Server to start the computation. When the Object-Server needs other servers to distribute the job, the Object-Server will send the request to TM through the SVR_OSB. TM will assign servers to OSB and OSB will send the information to the Object-Server. Then the Object-Server can distribute the job to the child Object-Servers. After the Child Object-Servers finish the computation, they will send the results to the parent Server. And then Object-Server will display the result to the clients through OSB.

2.2.2 Spider II Product Perspective (New System)

2.2.2.1 System Interfaces

The Spider II will keep the same major components as Spider I. Figure 2.2 shows the System Interface of the Spider System. The description of each component is described in section 2.1.
2.2.2.2 User Interfaces

Spider II uses text mode display the user can just run Spider on Unix or Linux console.

2.2.2.3 Hardware Interfaces

The Spider server is a Pentium 200 server, and the server workstations are Pentium II computers in JB356, JB358 and JB 359 computer labs.

2.2.2.4 Software Interfaces

The software required to develop Spider II are following:

- Linux RedHat 6.2
- GNU C++ programming for Linux

2.2.2.5 Communication Interfaces
Ethernet 100 base network connection for Linux box is used. The BSD Socket interface is used for the communication of all server objects. In order to maintain stable communication and provide better speed, TCP is the only protocol used in the implementation.

2.2.2.6 Operations

The operations of the Spider II will be following:

First, start the Task Manager, Registry Server and their Mirror Server from server stations. Then Clients can request the distributed service.

Spider II will only run during scheduled periods or by request. It will not run 24 hours a day, 7 days a week.

2.2.3 Product Functions

The different functions of Spider II compared with Spider are:

2.2.3.1 Registry Server and Load Balancing

The function of the Registry is to provide a list of available Object-Servers to OSB and Task Manager. In Spider, Registry gets the available Object-Servers' information from a file, and the information is updated by hand. If there is one server down, Registry server will not know this until the superuser change the server list file. In Spider II, the Registry Server will auto-detect Object-
Servers whether they are up or down and check the percentage of utilization of the CPU usage. After the detection, Registry will have a list of available servers and the priority of every Object-Server that the Registry Server suggests Task Manager to use. Then Spider will update the information list in the Task Manager.

2.2.3.2 The Mirror Registry Server

The Registry Server plays an important role in the distributed computation service. We do not want the Registry Server to go down during the computation. So in Spider II, we added a new mirror Registry Server. If the Registry Server goes down during the computation, the mirror will pick up the duty.

2.2.3.3 Ease of Adding New Service

Spider II will be designed in the object-oriented paradigm, so it will be easy to add new functions or services in the future. Some utility programs can be reused as Object Engines. Developers just need to know the design patterns and use the Object Engines to write a new distributed application, see Figure 2.3
2.2.3.4 Reduced Communication Protocols

Spider II reduces the communication overhead from 14 steps in Spider to 8 steps. The protocol of the distributed computation is shown in Figure 2.4.
When a client requests a service, OSB will ask the Task Manager for the task ID and the server objects that the client will require. Task Server will return the task ID and a list of object-servers that the OSB can use. Then OSB will distribute the job to the object-servers. After the object-servers finish the computation, OSB will collect the result and send back to the client. At the same time, OSB will return the task ID to the Task Server. Registry Server will send a list of available servers to Task Manager every
time it has an update. Both Task and Registry Server have mirrors to keep a copy of the information.

Shown on Figure 2.5 is the Use Case diagram of Spider II:

Figure 2.5. Use Case Diagram

2.2.4 User Characteristics

The User of Spider system does not need to know about programming. On the other hand, the developers for the
application programs need to have experience with C++ programming under UNIX environment, understand network programming concepts and know object oriented paradigm in order to maintain Spider and add new functions to it.

2.2.5 Constraints

- Spider II will only run in the CS Network.
- The platform will only support UNIX or UNIX-like operating systems. Linux will be the primary environment during development.

2.3 Specific Requirements

This section will detail some of the new functions of Spider II and its user interface. Spider II uses text-based user interface. We specify each interface in following sections.

2.3.1 Registry Server

The major functions of Registry Server include detecting available Object-Servers, checking the CPU usage of the available Object-Servers, assign the priority to the hosts, and updating the information in the Task Server. Figure 2.6 shows the interface of the Registry Server.
There are two groups of data shown on Figure 2.6. The first group shows the Object-Servers’ name and their CPU idle percentage, i.e. ‘avhosts 0 => jb358-0 99’. This means that the available host’s hostname “jb358-0”, and its CPU idle percentage is 99%. The second group shows the hostname of the Object-Servers and its priority. The priority rules are shown in table 2.2.
Table 2.2. Priority Rules

<table>
<thead>
<tr>
<th>Priority</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Priority 0</td>
<td>CPU idle &gt; 50%</td>
</tr>
<tr>
<td>Priority 1</td>
<td>CPU idle &gt; 30%</td>
</tr>
<tr>
<td>Priority 2</td>
<td>CPU idle &lt; 30%</td>
</tr>
<tr>
<td>Priority 3</td>
<td>The Object-Server is used by Spider II</td>
</tr>
<tr>
<td>Priority 4</td>
<td>Object Server not Available</td>
</tr>
</tbody>
</table>

2.3.2 Task Manager

When the Task server starts, it will load the initial value of the hostname, priority, port number, and the Task ID. As shown in Figure 2.7, the first column is the hostname, the second column is the priority, the third column is the port number, and the fourth column is the Task ID.

Figure 2.7. Task Server: Initial Value

![CRT - jb358-0](image)

The initial priority of every host is 4, which means they are not available. The initial value for Task ID is 0, because no task is assigned yet. After the Registry server sends the information of available Object-Servers, the Task
Server will update the priority of each host, as shown in Figure 2.8.

Figure 2.8. Task server: Get Information from Registry

When a job is requested from the OSB, the Task Server will assign the host and Task ID. Then it will change the priority field to 3 and the Task ID is assigned to the ID number as assigned, this is shown on Figure 2.9.

Figure 2.9. Task Server: After the Task ID is Assigned
2.3.3 Load Balancing

Load balancing is handled by both Registry and Task Server. Registry Server detects the available Object-Servers and checks their CPU usage. Task Manager keeps the Object-Server information database, which includes hostname, priority, Task ID, and TCP port number.

When Registry Server starts, it will detect the available Object-Servers and check their CPU idle percentage. And Registry will assign the priority, according to the CPU idle percentage. Then Registry Server will update the information in Task Server. The Registry Server will check all Object-Servers once every two minutes. The priority rule is shown in Table 2.2.

When Task Server receives information from the Registry Server, it will update its own Object-Servers information database and the database in the Task Mirror. If an OSB requests some Object-Servers, Task Server will assign the Object-Servers, which have priority "0" and change the Object-Servers' priority from priority "0" to priority "3", which means it is in use, see Table 2.2. If priority 0 Object-Servers is not enough, Task Server will assign priority 1 Object-Servers and then priority 2, and so on.
2.3.4 Distributed Computing Service

In order to utilize the distributed computing feature of Spider II, we chose three distributed computing services: They are the Distributed Quick Sort, Distributed Prime Number Search, and Distributed Matrix Multiplication. These applications are used for measure the performance of Spider II. Now, we will define the interface of each service.

2.3.4.1 Distributed Quick Sort

The users need to input the size of numbers the integer that will be sorted, and the number of Object-Servers. The quick sort client will generate random numbers to sort. During this process, users can see messages from OSB to indicate which Object Server are used for this service, and how much data are received by each Object Servers. When the service is done, the client will display the processing time of this service. Because of the large amount of data to be sorted (could be about 1 million numbers), we will not show the result on the screen, but the users will see pages and pages of sorted numbers scrolling on the screen.
2.3.4.2 Distributed Prime Number Search

Prime Number Search is to search for all prime numbers in a range of integers. In Distributed Prime Number Search, users can input the beginning and end number of the range or just the ending number. If users just input the ending number, the prime search will start from 1. Users will also need to input the number of Object-Servers stations that will be used. Just like Distributed Quick Sort, OSB will report the status of the processing information. After the service is done, the processing time will be shown on the screen.

2.3.4.3 Distributed Matrix Multiplication

For Distributed Matrix Multiplication, users will input the size of the matrices to be multiplied. The Distributed Matrix Multiplication will require nine Object Servers to run this service, because this is the most effective way to do the distributed matrix multiplication according to A.Geist[3]. The algorithm of the Distributed Matrix Multiplication will be specified in Chapter Three.
CHAPTER THREE: DESIGN OF SPIDER II

The software development used in implementing Spider II is object-oriented approach. Also, the design methodology uses the Unified Modeling Language (UML). UML is an industry-standard language for OO design of a software system. It simplifies the complex process of software design, by making a “blueprint” for the software construction. The pseudo-code algorithm, the architecture, and the detailed design of Spider II will be discussed.

In this Chapter, we will first look at the architecture of Spider II. Then, we will discuss the detailed design of each component or object in the architecture.

3.1 Architecture of Spider II

The architecture of Spider II is derived from the System Interface (Figure 2.3), and the protocols of the distributed computation (Figure 2.5). In this section, we will show the physical and logical design of the system.

3.1.1 System Design

The UML deployment diagram shows the physical components and the physical communication protocols between the components of the software system. Figure 3.1 shows the system design of Spider II.
When a Client Station requests a distributed service, OSB will be started in the local machine. OSB then communicates to the Task Server by using TCP Socket. It will also distribute data to object-servers using TCP. TCP protocol is used between Registry Server and Task Server.

**Figure 3.1. System Design of Spider II**

3.1.2 Software Architecture of Spider II

A UML class diagram is a graph of all the class elements and components of a software system connected by their various static relationships. Figure 3.2 shows the major classes or components in the architecture of Spider II.
The Socket class is the most important class in the Spider II design. It is used to build the communication protocols between different servers. Registry Servers, Task Server, and OSB need to instantiate the Socket class in order to build the communication protocols between them.

Registry class is the main class for Registry Server. It handles the tracking of available Object-Servers and performs load balancing. Task class is the main class for Task Manager, which manage the Object-Server usage. When we develop a distributed computing service using Spider II, we need to have client side classes, which receives data and organizes the data for distribution. And we need server side classes to do the computation on Object-Servers.

Client side application need to instantiate OSB class, because OSB takes care of communication and distribution functions for client side program. Server side classes need to instantiate osb_svr class, which receives data from OSB, and returns data to OSB when Object-Server finishes the computation.
3.1.3 Component Architecture of Spider II

A UML component diagram is a graph of components connected by dependency relationships. Components may also be connected to components by physical containment representing composition relationships. Figure 3.3 shows the major components and their relationship in the design of Spider II.

The design of Spider II consists of three major components: Distributed Service, Communication Programs, and Object-Server Management. Each major component is the
collection of sub-components. Distributed Service contains the user interface, and the computation programs for Spider II. Communication Program provides the communication protocol for Spider II. Object-Server Management provides load balancing and manages the usage of Object-Servers.

Figure 3.3. Component Architecture of Spider II

3.2 Detailed Design

The detailed design of the Spider II will be discussed in the following sections.

3.2.1 Task Manager

Figure 3.4 shows the class diagram of the Task Manager. Task Manager has two parts: Task Server and its Mirror Server. Task is the main class for Task Server and
T_mirror is for the Mirror Server. The Task class contains the functions of the Task Server. T_mirror class needs aggregate Task class, so Mirror Server can perform the Task Server’s job when the Task Server goes down or crashes. Socket class is used for the communication between Task Server and Mirror server and the Task Server and OSB, so both Task and T_mirror need to instantiate Socket.

Figure 3.4. Task Manager Class Diagram

3.2.2 Registry Server

Figure 3.5 shows the class diagram of the Registry Server. It has two important classes: Registry class and reg_test class. Registry class keeps the object servers
information and connects the Task Server. `reg_test` tests the available servers, and the `CPUusage` class is instantiated by `reg_test` to test the CPU usage.

**Figure 3.5. Registry Server**

3.2.3 Object Service Broker (OSB)

The design of Object Service Broker includes two major classes: `OSB` and `osb_srv`. `OSB` is the main class for Object Service Broker. `osb_srv` is the remote site class running on Object-Servers. When a job needs to be distributed `OSB` will wake up `osb_srv` so `OSB` can distribute the data to the Object-Servers. Figure 3.6 shows the class diagram for `OSB`.

31
3.2.4 Distributed Computing Service Applications

There are three applications currently implemented in Spider II system. They are Distributed Quick Sort, Distributed Prime Number Search, and Distributed Matrix Multiplication. Since Spider II has Task Manager, Registry Server, and OSB designed in Object-Oriented approach. Designing a distributed application becomes quite easy. Figure 3.7 is a class diagram for one of the application, i.e., Distributed Prime Number Search. An application needs two classes: one is running on the client side and the other is running on the Object-Servers. The basic function in the client side class is to give data to OSB, tell OSB
the number of Object-Servers needed to perform the job, and how the data will be distributed. The function Object-Server side class is receiving data and performing the computation.

Figure 3.7. Application: Prime Number Search

Now, let us discuss the distributed algorithm of each application.

3.2.4.1 Distributed Quick Sort

The quick sort algorithm provides a fast and efficient way to sort data by recursively calling itself. The algorithm is expressed as follow:
Array A[1..N] is to be sorted. 

procedure QSORT(lo, hi)
if lo < hi
    then [ i ← lo; j ← hi+1; K ← K_m
        loop
            repeat i ← i+1 until K_i ≥ K;
            repeat i ← i+1 until K_i ≥ K;
            if i < j
                then swap (A[i], A[j])
                else exit
        forever
        swap (A[m], A[j])
        call QSORT(lo, m-1)
        call QSORT (m+1, hi)]
end QSORT

The average computing time for quick sort is $O(N \log_2 N)$.

The algorithm of Distributed Quick Sort is to divide an array of numbers and send them to the Object-Servers to perform quick sort. After finishing the sort on each Object-Server, they will send the sorted array to the client station, and the client will do the final merge sort. Figure 3.8 shows the algorithm of the distributed quick sort.
3.2.4.2 Distributed Prime Number Search

The Prime Number algorithm is expressed as follow:

```plaintext
procedure prime_number_search(begin_num, end_num)
/* "begin_num" is the beginning number of the sequential number. "end_num" is the ending number of the sequential number, Array "arr" store the result, "index" is the array counter. */

index ← 0
for (N = begin_num to end_num, N ← N+1)
    if N <= 3
        then arr[index] ← N
            index ← index+1
    end if
    else if ((N mod 2) ≤ 0 and N > 3)
        then
            while (j ≤ (N/2))
                j ← j+2
                if ((N mod j) = 0)
                    then break;
                else if (j ≥ (N/2))
                    then arr[index]=N
                        index ← index+1
                end if
            end while

end if
```

35
end while loop
end if
end for loop
end prime_number_search

The average computing time for the prime number search is $O(N^2)$.

From the prime number search algorithm, we know that when the number becomes bigger, it will take more time to search for the prime number. So we design the algorithm of the Distributed Prime Number Search as shown in Figure 3.9. If a user needs $N$ object-servers for this job, the client will divide the data to $2*N$ sections, and send section 0 and section $N$ to Object-Server 1, section 1 and section $N-1$ to Object-Server 2 and so on. After the computation, Object-Servers will send their results back to the client and then the client will merge the result.
3.2.4.3 Distributed Matrix Multiplication

The algorithm of the Matrix Multiplication is expressed as follow:

\[ M_3[1..n][1..n] = M_1[1..n][1..n] \times M_2[1..n][1..n] \]

procedure Matrix_Multiplication
loop
  repeat i \leftarrow i+1 until i \geq n
  repeat j \leftarrow j+1 until j \geq n
  repeat k \leftarrow k+1 until k \geq n
    \[ M_3[i][j] = M_3[i][j] + M_1[i][k] \times M_2[k][j] \];
  forever
end Matrix_Multiplication

The average computing time for Matrix Multiplication is \( O(N^3) \).
The algorithm of the Distributed Matrix Multiplication that we used is a sub-matrix algorithm [3] as shown as follows:

Figure 3.10. Sub-Matrix Algorithm

\[
\begin{array}{ccc}
\text{A} & \times & \text{B} \\
00 & 01 & 02 \\
10 & 11 & 12 \\
20 & 21 & 22 \\
\end{array}
\]

\[
\begin{array}{ccc}
\text{C} & = & \\
00 & 01 & 02 \\
10 & 11 & 12 \\
20 & 21 & 22 \\
\end{array}
\]

\[
C_{00} = A_{00} \times B_{00} + A_{01} \times B_{10} + A_{02} \times B_{20}
\]
\[
C_{01} = A_{00} \times B_{01} + A_{01} \times B_{11} + A_{02} \times B_{21}
\]
\[
C_{02} = A_{00} \times B_{02} + A_{01} \times B_{12} + A_{02} \times B_{22}
\]
\[
C_{10} = A_{10} \times B_{00} + A_{11} \times B_{10} + A_{12} \times B_{20}
\]
\[
C_{11} = A_{10} \times B_{01} + A_{11} \times B_{11} + A_{12} \times B_{21}
\]
\[
C_{12} = A_{10} \times B_{02} + A_{11} \times B_{12} + A_{12} \times B_{22}
\]
\[
C_{20} = A_{20} \times B_{00} + A_{21} \times B_{10} + A_{22} \times B_{20}
\]
\[
C_{21} = A_{20} \times B_{01} + A_{21} \times B_{11} + A_{22} \times B_{21}
\]
\[
C_{22} = A_{20} \times B_{02} + A_{21} \times B_{12} + A_{22} \times B_{22}
\]

As shown in Figure 3.10a, a matrix can be divided into 9 sub-matrices, and the multiplication is shown in Figure 3.10b. From the Sub-Matrix Algorithm, we can develop the Distributed Matrix Multiplication as shown in Figure 3.11. In order to reduce the communication overhead, we send
$A_{00}, B_{00}, B_{01},$ and $B_{02}$ to Object-Server 1. Then we can get $A_{00} \times B_{00}, A_{00} \times B_{01},$ and $A_{00} \times B_{02}$. They are partial results of the answer $C_{00}, C_{01},$ and $C_{02}$. We send $A_{01}, B_{10}, B_{11},$ and $B_{12}$ to Object-Server 2 and $A_{02}, B_{20}, B_{21},$ and $B_{22}$ to Object-Server 3. We add the result that came back from Object-Server 1, Object-Server 2 and Object-Server 3. Then we can get the complete results $C_{00}, C_{01},$ and $C_{02}$, See Figure 3.12.

Figure 3.11. Distributed Matrix Multiplication Algorithm

Figure 3.12. The Final Process in Client

$$
C_{00} = A_{00} \times B_{00} + A_{01} \times B_{10} + A_{02} \times B_{20} \\
C_{01} = A_{00} \times B_{01} + A_{01} \times B_{11} + A_{02} \times B_{21} \\
C_{02} = A_{00} \times B_{02} + A_{01} \times B_{12} + A_{02} \times B_{22}
$$

Results form Object Server 1
Results form Object Server 2
Results form Object Server 3
CHAPTER FOUR: SYSTEM VALIDATION

The system validation is the process of conducting critical test and the results are evaluated against design specifications and intended functionality. The purpose of the system validation is to assure the quality and reliability of Spider II.

A series of different schemes are undertaken to test Spider II. In this chapter, each test scheme is presented in detail.

4.1 Unit Test

Unit test is the initial step in the software testing process. The unit test focuses on the smallest functional unit of design. The unit test of Spider II consists of testing the functions of each component if it works as expected. The results of the unit test is shown in Table 4.1:

Table 4.1. Unit Test

<table>
<thead>
<tr>
<th>Forms</th>
<th>Tests Performed</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registry Server</td>
<td>• Verify Registry Server reads the hostname of Object-Servers from data-file</td>
<td>Pass</td>
</tr>
<tr>
<td></td>
<td>• Verify Registry server detecting the Object-Servers.</td>
<td></td>
</tr>
<tr>
<td>Registry Server</td>
<td>• Verify Registry Server update Object-Servers information to</td>
<td></td>
</tr>
</tbody>
</table>
| Task Server          | Registry Server Daemon | • Verify Registry Server detects available Object-Servers in every two minutes  
|                     | Registry Mirror Daemon | • Verify the Mirror Server does not detect Object-Servers in the first minutes  
|                     | Task Manager           | • Verify Task-Mirror starts and sets default information for Object-Servers including hostname, priority = 4, default port number, and Task ID = 0  
| Task mirror start   | Task Server start      | • Verify Task Server starts and sets default information as above  
| Receive data from Registry server | Receive request from OSB | • Verify Task Server receive new Object-Servers' information from Registry Server  
| Receive return from OSB |                     | • Verify Task ID is assigned  
|                     | Pass                   | • Check the host assignment is from priority 0.  
|                     | Pass                   | • Verify new information is updated  
|                     | Pass                   | • Verify the information is updated to Task Mirror  
|                     | Pass                   | • Verify Task ID receive from the OSB  
|                     | Pass                   | • Check Task ID is changed back to default for those hosts  
|                     | Pass                   |  

41
<table>
<thead>
<tr>
<th>Task</th>
<th>Requirement</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mutli-Tasking</td>
<td>• Verify two different OSB request from different workstations.</td>
<td>Pass</td>
</tr>
<tr>
<td>Object Service Broker</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Communication with Task Manager</td>
<td>• Check Task ID</td>
<td>Pass</td>
</tr>
<tr>
<td></td>
<td>• Verify the hostname list receives from Task Server</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Check if the Task Server down, OSB sends request to Task Mirror</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Verify return Task ID to Task Server after finish the job</td>
<td></td>
</tr>
<tr>
<td>Communication and Distribution</td>
<td>• Verify OSB start Object-Servers</td>
<td>Pass</td>
</tr>
<tr>
<td></td>
<td>• Verify Object-Servers received data as same as OSB sent</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Verify Object-servers return data to OSB after the computation</td>
<td></td>
</tr>
<tr>
<td>Application Test</td>
<td>• Verify the result from each Object-Server are accurate</td>
<td>Pass</td>
</tr>
<tr>
<td>Distributed Prime Number Search</td>
<td>• Verify the result amount different size of data are accurate</td>
<td></td>
</tr>
<tr>
<td>Distributed Matrix Multiplication</td>
<td>• Verify the result from each Object-Server are accurate</td>
<td>Pass</td>
</tr>
<tr>
<td></td>
<td>• Verify the result amount different size of data are accurate</td>
<td></td>
</tr>
<tr>
<td>Distributed Quick Sort</td>
<td>• Verify the result from each Object-Server are accurate</td>
<td>Pass</td>
</tr>
<tr>
<td></td>
<td>• Verify the result amount different size of data are accurate</td>
<td></td>
</tr>
</tbody>
</table>
4.2 Load Balancing Test

Load balancing is one of the important improvements from Spider I, and it helps Spider II provide better performance. Load balancing ensures the good performance of the system. The results of the load-balancing test is shown in Table 4.2:

<table>
<thead>
<tr>
<th>Forms</th>
<th>Tests Performed</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registry server check the CPU idle</td>
<td>• Verify Registry Server detect CPU idle percentage of Object-Server</td>
<td>Pass</td>
</tr>
<tr>
<td></td>
<td>• Check the Registry Server convert CPU idle percentage to the priority number</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Verify Registry Server sends available Object-Servers' hostname and priority to Task Server</td>
<td></td>
</tr>
<tr>
<td>Registry assign priorities</td>
<td>• Check convert CPU idle percentage to the priority number as follow:</td>
<td>Pass</td>
</tr>
<tr>
<td></td>
<td>more than 50% -&gt; priority 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>more than 30% -&gt; priority 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>less than 30% -&gt; priority 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• If an Object-Server is down, the priority is 4</td>
<td></td>
</tr>
<tr>
<td>Update to Task Server</td>
<td>• Verify Registry the now Object-Servers information to Task Server</td>
<td>Pass</td>
</tr>
<tr>
<td>Task Server assign order</td>
<td>• Verify Object-Servers are assigned by the order of priority 0, priority 1 and priority 2</td>
<td>Pass</td>
</tr>
</tbody>
</table>
4.3 Fault Tolerance Test

Fault Tolerance is also a very important issue in order to provide a stable system. A series of fault tolerance tests are taken to ensure the stability of Spider II. The results of the fault tolerance test is shown in Table 4.3

Table 4.3. Fault Tolerance Test

<table>
<thead>
<tr>
<th>Forms</th>
<th>Tests Performed</th>
<th>Results</th>
</tr>
</thead>
</table>
| Shutdown Task Server without any running task | • Shutdown the Task Server, verify the Registry Server sends data to Task Mirror  
• Check a new request from OSB will directly go to Task Mirror | Pass    |
| Shutdown Task Server with running tasks | • Verify OSB go to the Task Mirror when OSB can not find the Task Server  
• Verify Registry Server directly go to Task Mirror when it update the available Object-Server information | Pass    |
| Restart Task Server          | • Verify Task Server copy information from Task Mirror when it restart  
• Verify Registry Server and OSB communicate with Task Server again | Pass    |
4.4 System Test

System test of Spider II consists of testing the maximum data size that the system can process in each application. We first use small amount of data, and then increase the data size until the system cannot handle it anymore. The result is shown in Table 4.4:

Table 4.4. Maximum Data Size of Spider II Applications

<table>
<thead>
<tr>
<th>Applications</th>
<th>Maximum Data Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distributed Matrix Multination</td>
<td>900x900</td>
</tr>
<tr>
<td>Distributed Prime Number Search</td>
<td>2,000,000</td>
</tr>
<tr>
<td>Distributed Quick sort</td>
<td>1,000,000</td>
</tr>
</tbody>
</table>

4.5 System Performance Test

In this section, we evaluate the performance of the Spider II. We compare the performance of Spider II with high-end and low-end sequential computers.

4.5.1 Testing Environments

High-end sequential computer:

Dell PowerEdge 4400
Intel Pentium III Xeon 800 MHz CPU
1 GB memory

Middle-level sequential computer:

AMD Athlon 700 CPU
64 MB memory

Low-end sequential computer:

NEC PowerMate 1800
4.5.2 Testing Method

We use different sizes of data to test the applications running on the sequential computers and Spider II. We then record the time when each system finished the job. Testing sizes of Quick Sort range from 100,000 to 1,000,000. The testing matrix sizes for Matrix-Multiplication are from 100x100 to 900x900. The test data sizes of Prime Number Search are from 50,000 to 2,000,000.

4.5.3 Testing Results

4.5.3.1 Matrix Multiplication

In this test, Spider II uses 9 Object-Servers. The results are shown in Table 4.5 and Figure 4.1:

Table 4.5. Comparison of Matrix Multiplication

<table>
<thead>
<tr>
<th>Matrix Multiplication</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spider II</td>
</tr>
<tr>
<td>-----------</td>
</tr>
<tr>
<td>100 x 100</td>
</tr>
<tr>
<td>200 x 200</td>
</tr>
<tr>
<td>300 x 300</td>
</tr>
<tr>
<td>400 x 400</td>
</tr>
<tr>
<td>500 x 500</td>
</tr>
</tbody>
</table>
In Figure 4.1 and Table 4.5, when the matrix size is small, i.e. 100x100 or 200x200, all the sequential machines are faster than Spider II. When the matrix size increases to 300x300, Spider II is faster than Pentium II 350 computer. When the matrix size increases to 400x400, Spider II is faster than AMD Athlon 700. When the Matrix size is
increased to 500×500, Spider II is faster than the high-end Pentium III Xeon 800 Server.

It is found that the maximum testing matrix size for sequential computers is 835×835. In the Matrix Multiplication program for sequential computer, we can only define the matrix size of up to 835×835. The sequential computes define the matrix size larger than 835×835, it will run into core dump during the execution. In Table 4.5, we put "core dumped" in the row of 900×900. The maximum matrix size that Spider II can process is 900×900. This test also shows that Spider II can process larger data size that sequential computer cannot handle.

4.5.3.2 Prime Number Search

In this test Spider II uses 8 Object-Servers. The results are shown in Table 4.6 and Figure 4.2:

Table 4.6. Comparison of Prime Number Search

<table>
<thead>
<tr>
<th></th>
<th>Prime Number Search</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Spider II</td>
</tr>
<tr>
<td>50,000</td>
<td>4</td>
</tr>
<tr>
<td>80,000</td>
<td>6</td>
</tr>
<tr>
<td>100,000</td>
<td>7</td>
</tr>
<tr>
<td>200,000</td>
<td>10</td>
</tr>
<tr>
<td>300,000</td>
<td>18</td>
</tr>
<tr>
<td>400,000</td>
<td>28</td>
</tr>
<tr>
<td>500,000</td>
<td>42</td>
</tr>
<tr>
<td>600,000</td>
<td>62</td>
</tr>
</tbody>
</table>
From Table 4.6 and Figure 4.2, we know that for data size under 100,000, the performance of Spider II and sequential computers are very close. But when the data size increases, we can see that Spider II runs much faster than any of the sequential machines. In Table 4.6, we put "Too Long" in some entries, because we waited more than 16
minutes for the sequential computers to finish jbos, but they are still computing, so we terminate the jobs.

4.5.3.3 Quick Sort

In this test Spider II uses 8 Object-Servers. The results are shown in Table 4.7 and Figure 4.3:

Table 4.7. Comparison of Quick Sort

<table>
<thead>
<tr>
<th>Spider II</th>
<th>Pentium II 350</th>
<th>AMD Athlon 700</th>
<th>Pentium III Xeon 800</th>
</tr>
</thead>
<tbody>
<tr>
<td>100000</td>
<td>3</td>
<td>0.2</td>
<td>0.1</td>
</tr>
<tr>
<td>200000</td>
<td>4</td>
<td>0.4</td>
<td>0.3</td>
</tr>
<tr>
<td>300000</td>
<td>4</td>
<td>0.7</td>
<td>0.5</td>
</tr>
<tr>
<td>400000</td>
<td>4</td>
<td>0.9</td>
<td>0.7</td>
</tr>
<tr>
<td>500000</td>
<td>5</td>
<td>1.2</td>
<td>1</td>
</tr>
<tr>
<td>600000</td>
<td>5</td>
<td>1.5</td>
<td>1.3</td>
</tr>
<tr>
<td>700000</td>
<td>6</td>
<td>1.9</td>
<td>1.5</td>
</tr>
<tr>
<td>800000</td>
<td>6</td>
<td>2.2</td>
<td>1.8</td>
</tr>
<tr>
<td>900000</td>
<td>7</td>
<td>2.5</td>
<td>2</td>
</tr>
<tr>
<td>1000000</td>
<td>7</td>
<td>3</td>
<td>2.4</td>
</tr>
</tbody>
</table>
From the Table 4.7 and Figure 4.3, we know that no matter how big the data size is, all the sequential machines have better performance than Spider II.

4.5.4 Performance Analysis

From the results in section 4.5.3, we see that in the Matrix Multiplication and Prime Number Search, Spider II has better performance when the data size is large. But in Quick Sort, Spider II is slower than any sequential machines. We conclude with two reasons:

First, the computing time of the Matrix Multiplication is $O(N^3)$ and Prime Number Search is $O(N^2)$. These two
algorithms require complex computation, and they really need more computation power when the CPU needs to process large amount of data. These two algorithms can really take advantage on distributed computing, because it separates large data into small amount of data and distributed to several computers to process, which saves lots of time. On the other hand, the computing time for Quick Sort is $O(N \log_2 N)$, which is a very fast algorithm. It is very hard to take advantage on distributed computing and 1,000,000 numbers are not large for Quick Sort.

Second, the Distributed Quick Sort algorithm requires the final sort on the client stations; see Figure 3.8 in Section 3.2.4.1. This process takes time too. But from Table 4.7, we can see that the data size increase 10 times, from 100,000 to 1,000,000, the execution time of Spider II increases about two times only. In the sequential machines, the execution time increases more then 10 times. Therefore, in a extremely large data size, distributed quick Sort for Spider II may have better performance than sequential machines. This was not tested in this project.
CHAPTER FIVE: MAINTENANCE MANUAL

Maintenance is an important process to keep the software up and running. Since the Spider Project is an ongoing project, maintenance manual also helps in future development. Maintenance of Spider II consists of two sections: first, installation and configuration of Spider II, and second, system monitoring and error checking.

5.1 System Requirement

Spider II has only been tested on RedHat Linux, and kernel 2.2.x. If you use other Linux distributions with kernel 2.2.x, Spider II should be fine on your system. But kernel 2.2.14 has some problems with communication and security. If you are using RedHat Linux 6.2 or any Linux with kernel 2.2.14, we strongly suggest that you upgrade your kernel to 2.2.16 or up.

Spider II also requires GNU project C++ Compiler, "g++", to compile the system.

5.2 Installation and Configuration

First step of installing Spider II is downloading the source code. There are two places to download the source code: First one is on the Spider Server, and the file is located on:

ftp://spider.csci.csusb.edu/pub/spider_2_src.tar.gz
The second is on the Network File Server (NFS) of the CS Network, which is located on:

```
/share/spider/spider_2_src.tar.gz
```

If you have a computer science network account of CSUSB, you can just copy the file to your own directory. After you downloaded the file, you need to unpack the file. In the shell prompt, type:

```
tar zxvf spider_2_src.tar.gz
```

Then, you will see the directory spider_II.

The next step is compiling Spider II. But first, you need to configure the file profile.h. This file is a header file, and it sets the environment of the Spider II system. The file is in the directory spider_II/src/. Some needs to be changed, an example is shown in Table 5.1:

Table 5.1. "profile.h" file

```
#define Spiderpath "./share/spider/spider_II/bin"
#define DATA_FILE "./share/spider/spider_II/reghost.conf"
#define HOSTFILE "./share/spider/spider_II/ports.conf"
#define Task_Flag "./share/spider/spider_II/tflag.dat"
#define Task_Server "jb358-0"
#define Task_Mirror "jb358-1"
#define T_manager_port 8200
#define T_mirror_port 8201
#define Reg_port 7200
```
You need to change the path of "Spiderpath", "DATA_FILE", "HOSTFILE", and "Task_Flag". The "Task_Server" is the hostname of the computer, which you like to run your Task Server. "Task_Mirror" is the hostname of the Task Mirror. "T_manager_port" is the TCP port number used by the Task Manager and "T_mirror_port" is the TCP port number used by the Task Mirror. "Reg_port" is the TCP port number used by the Registry Server.

After you changed these values, you can use 'make all' to compile the system, which will make all the binary files and commands for Spider II, and those files will be in the directory bin. The commands of Spider II are shown in Table 5.2

Table 5.2. Commands of Spider II

<table>
<thead>
<tr>
<th>Task</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task</td>
<td>Task Server</td>
</tr>
<tr>
<td>Tm</td>
<td>Task Mirror</td>
</tr>
<tr>
<td>reg_d</td>
<td>Registry Server</td>
</tr>
<tr>
<td>reg_mirror_d</td>
<td>Registry Mirror</td>
</tr>
<tr>
<td>matrix_p</td>
<td>Distributed Matrix Multiplication</td>
</tr>
<tr>
<td>Prime</td>
<td>Distributed Prime Search</td>
</tr>
<tr>
<td>Qsort</td>
<td>Distributed Quick Sort</td>
</tr>
</tbody>
</table>

There are some other binary files that are not used by a user but are used by Spider II systems: "reg" is used by Registry Server and Mirror. "matrix_srv", "ps_srv", and "qs_srv" are the applications that runs on Object-Servers
and used by Distributed Matrix Multiplication, Distributed Prime Number Search, and Distributed Quick Sort applications.

We suggest you set the environment variable SpiderPATH to the path where the commands of Spider II resides, i.e. /share/spider/spider_II/bin.

If your shell is Korn shell or Bourne shell, add the following lines in .profile or .bash_profile file:

```bash
export SpiderPATH=/share/spider/spider_II/bin
PATH=$PATH:$SpiderPATH
export PATH
```

If your shell is C shell, add these lines to .cshrc

```bash
setenv SpiderPATH=/share/spider/spider_II/bin
PATH=$PATH:$SpiderPATH
```

5.3 Running Spider II system

To start Spider II, you need to start Task Server (task) and Task mirror (tm) on the computers that you set in the profile.h file. This will start Registry Server and Registry Mirror. After the Registry Server tests the available Object-Servers and update the available Object-Servers information in the Task Server, you can run the applications.

5.4 Adding a New Application
Since Spider II used the object-oriented paradigm, adding a new system in Spider II is very easy.

5.4.1 Client Program

First, you need to prepare your data. You need to put your data into an array. Then there are some necessary OSB functions that needs to be called.

```c
void spider_spawn(char *service, int num):
spider_spawn: start OSB, and OSB will ask the Task Server for the object servers.
Parameter:
char *service - the file name of the server application.
int num - number of the object servers request from task server.
```

```c
void send_data(int *data, int size):
send_data: distributed data functions
parameter:
int *data - the data array that needs to be distributed
int num - the size of the data array.
```

```c
int get_return(int *data):
get_return: the result from object servers, this function will be returned.
parameter:
```
int *data - the array which store the result from object servers.

5.4.2 Server Program

First, call osb_svr to get data from the client.

int get_data(int *data)

get_data: get data from client. This function returns the size of the data array.

parameter:

int *size - the data array received from the client

Then you need to have some functions to do the necessary computations.

And then, return data to client by calling this function.

void return_data(int *data, int datasize)

return_data: return data to client.

Parameter:

int *data - the data array containing the result after computation.

int datasize - the size of the result data size.

Finally, you compile the client and server, then you get a new application.
CHAPTER SIX: FUTURE DIRECTIONS AND CONCLUSIONS

The Spider II distributed computing system was presented in this project, and is intended to provide a distributed computing environment in the CS network of CSUSB. Spider II provides a fundamental base of any additional and improved implementations of the distributed computing applications. The following are suggestions for future implementations:

- Java-Based Interface Graphic Management Console: A graphic interface management console can provide a friendly GUI to users and the Spider II system administrator. The Java-based user interface can allow users to access Spider II from any platforms where there is a Web browser. The functions will include:
  - Provide the different level of authorization such as administrator and user.
  - Spider II can start and stop Task Server, Task Mirror, Registry Server, and Registry Mirror from the management console.
  - Monitoring the usage of the Object-Servers.
Run applications from the GUI and view the progress of the distributed computation process.

Intelligent Distributed Agents: Intelligent distributed agent is a new concept for the distributed computing. An Intelligent Agent can move from station to station. The major function of the Intelligent Agent is to detect the availability of Object-Servers and distribute data from client to the Object-Servers. When a client starts a distributed task, the Intelligent Agent will start from the client station and will move from Object-Server to Object-Server in order to collect the needed Object-Servers for this distributed task. The Intelligent Agent is smart and it can make decisions, such as how many Object-Servers will it use for this task and what kind of distributed computation algorithm it will use. Then it will distribute data to the Object-Servers and perform the computation. It will also help the client to receive results from the Object-Servers. After the task is finished, the Intelligent Agent will terminate itself. The Intelligent Agent will replace the Object Service Broker, Task Manager,
and Registry Server of Spider II. The advantage of the Intelligent Agent is mobility, prevention of system failure and elimination of disruption of service.

Java Pre-processor: The Pre-processor can help a user write applications for the Spider system. A programmer will write an application in Java. The pre-processor will compile the Java program and produce a set of independent Java components of the original program that can be processed in a distributed and parallel manner in Spider.

In conclusion, Spider II distributed computing system project is able to accomplished the following:

1. Spider II provides better functionalities than Spider I, such as load balancing and fault tolerance. Spider II also improves its computational power. In Spider I, the Distributed Quick Sort can sort only up to 100,000 integers. But in Spider II, the Distributed Quick Sort is able to sort up to 1,000,000 numbers. For the Distributed Matrix Multiplication, Spider I can only process a matrix size of up to $300 \times 300$, but Spider II can process up to
900x900 matrix. Spider II reduces the communication protocols thus improving its performance than Spider I.

2. Spider II is able to demonstrate that for complex computation, distributed computing using low-end workstations can provide better performance and computing power than a high-end server.

3. Spider II can provide an environment to study distributed network computation for the undergraduate and graduate operating system or data communication course in Computer Science.

Moreover, the experiences in implementing this project provided valuable knowledge of the distributed and parallel computing for me.
APPENDIX A: GLOSSARY

ATM
Asynchronous Transfer Mode, a high speed, scalable bandwidth network technology. It provides 155 Mbps transmission rate.

Client
The client is a workstation, which requests a service from the Spider system.

Client/Server Model
Architecture in which one computer can get information from another. The client is the computer that asks for access to data, software, or services. The server, which can be anything from a personal computer to a mainframe, supplies the requested data or services for the client.

Daemon
A UNIX program that runs continuously in the background, until it is activated by a particular event. This word is often used to refer to programs that handle email. The word daemon is Greek for "an attendant"
Distributed System

A distributed system is a collection of computers which run their own operating systems or a distributed operating system without having a global memory or a single clock, and computers communicate with each other by exchanging messages over a network.

Fault Tolerance

Fault tolerance refers to the ability of a computer or system to continue operations when a server error or failure occurs.

GUI

Graphical User Interface, an interface that has image as well as words on the screen.

JAVA

A cross-platform programming language from Sun Microsystems that can be used to create animations and interactive features on World Wide Web pages. Java programs are embedded into HTML documents.
Load Balancing: Distributing processing and communications activity evenly across a computer network so that no single device is overwhelmed.

OSB: Object service broker (OSB) is the central component of the Spider System to handle the communication between all objects in the system, regardless of their location, platform or implementation.

Protocol: A set of rules that regulate the way data is transmitted between computers.

Registry Server: The server provides the information of the available servers.

RPC: Remote Procedure Call. RPC mechanism provides transparency and efficiency for programs in the distribute system. Network messages are transferred to the remote system using either UPC, or TCP protocol.

Object-Servers: Network computers actually perform the computation during the distributed
| **Socket** | A communication between two computer processes on the same machine or different machines. On a network, sockets serve as endpoints for exchanging data between computers. Each socket has a socket address, which is a port number plus a network address. |
| **Spider Server** | The Web server which runs Java GUI daemon and provide Java applets for the client side of Spider System. |
| **Task Manager** | To manage the available servers for each task during computation in Spider System. |
| **TCP/IP** | The Transmission Control Protocol (TCP) / Internet Protocol (IP). These protocols were developed by DARPA to enable communication between different types of computers and computer networks. The Internet Protocol is a connectionless protocol, which |
provides packet routing. TCP is connection-oriented and provides reliable communication and multiplexing.

The Unified Modeling Language (UML) is a third-generation object-oriented modeling language for specifying, visualizing, and documenting the artifacts of an object-oriented system under development.
APPENDIX B: SOURCE CODE

//*****************************
//**** profile.h ****
//**** Header file for Spider II ****
//*****************************

#define MAXSTR 64
#define MAXBUFSIZE 5000
#define MARK '#'
#define DELM ':'
#define Spiderpath "/pool/major/grad/kwang/spider_II/bin/
#define DATA_FILE "/pool/major/grad/kwang/spider_II/ighost.conf"
#define HOSTFILE "/pool/major/grad/kwang/spider_II/ports.conf"
#define UPDATEFILE "/tmp/update_host.dat"
#define Task_Flag "/pool/major/grad/kwang/spider_II/tflag.dat"
#define Task_Server "jb358-0"
#define Task_Mirror "jb358-1"
#define T_manager_port 8200
#define T_mirror_port 8201
#define OSB_port 9220
#define Reg_port 7200
#define FROMLOCAL 1
#define FROMTASK 2
#define TASKINIT 3
#define FROMREG 4
#define FROMMRIR 5
#define OSBREQUEST 6
#define OSBRETURN 7

#define MAX 1000000
#define MAXPIPE 1000000
#define RSH "rsh"

#define T_Flag 0
#define M_Flag 1
// ********* task.h *********
// Header file for Task Server
// ***********************
#include <iostream.h>
#include <fstream.h>
#include <stdio.h>
#include <string.h>
#include "Socket.C"
#include "profile.h"
#include "util.C"

#define SYN_IO_S(x) (sck->sendint(x, &ok))
#define SYN_IO_R(x) (sck->recvint(x, &ok))
#define INT sizeof(int)
#define STRING (sizeof(char)*MAXSTR)
#define MAXHOST 100

class AV_HOST
{
    public:
        char name[MAXSTR];
        int priority;
};

class HOST
{
    public:
        char name[MAXSTR];
        char portnum[4];
        int priority;
        int id;
        int p_backup;
};

class TEMP
{
    public:
        char arr[MAXSTR];
};

class Task
{
    public:
        Task();
        ~Task();
        void waiting();
        void get_hostinfo();

    protected:
        HOST hosts[MAXHOST];
        AV_HOST avhosts[MAXHOST];
        TEMP temp[MAXHOST*4];
        char buf[MAXBUFSIZE];
        char local_host[MAXSTR];
char remote_host[MAXSTR];
int myport, sk, mirror_port;
char *t_manager, *t_mirror;
int count, pack_count;
int total_hosts_count;
int assign_id;

SOCKET *sck;
fstream file_fd;

void unpack(char *pk, int off);
void pack(char *pk);
void update_server_list();
int assign_hosts(int num);
void return_hosts(int num);
void save_info();
void unpack_hosts_info();
void pack_all_host_info();
void host_info_from_mirror();
void update_mirror();
delay();
void read_newinfo();
void reset();

/**
* task.obj.C
* Object Class for Task manager
* SPIDER II
*/
#include "task.h"

Task::Task()
{
  int i;
  for(i=0;i< MAXHOST;i++)
  {
    bzero(hosts[i].name, sizeof(hosts[i].name));
    bzero(hosts[i].portnum, sizeof(hosts[i].portnum));
    hosts[i].priority = 0;
    hosts[i].id = 0;
  } /* initial HOST */
  for (i=0; i < MAXHOST; i++)
    bzero(avhosts[i].name, sizeof(avhosts[i].name));

  assign_id = 0;
  total_hosts_count = 0;
  count =0;
  myport = T_manager_port;
  mirror_port = T_mirror_port;
t_mirror = Task_Mirror;
bzero(local_host, sizeof(local_host));
bzero(remote_host, sizeof(remote_host));
gethostname(local_host, MAXSTR);
printf("my hostname is %s\n", local_host);
}

Task::Task()
{
    delete sck;
}

void Task::unpack(char *out, int offset)
{
    char ch;
    int i=0, j=1;
    int k=0;
    if( j == offset) {
        while(buf[i] != DELM) {
            out[k++] = buf[i++];
        }
    }
    else {
        while(i < count) {
            while(buf[i++] != DELM);
            j++;
            if(j == offset) {
                while(buf[i] != DELM) {
                    out[k++] = buf[i++];
                }
            }
        }
    }
}

void Task::pack(char *pk)
{
    int i=0, len = strlen(pk);
    for(i=0;i<len;i++)
        buf[count++] = pk[i];
    buf[count++] = DELM;
}

void Task::waiting()
{
    int i, j, msg, num, datasize;
    pid_t childpid;
int cnt, newid, return_id, pipes[2];
int ok = 1;
int key = 1;

if(myport == 0) {
    fprintf(stderr, "Can't get T_manager port");
    exit(0);
}

sck = new SOCKET(myport, SOCK_STREAM);

if (sck ->sev_callsock() >=0)
{
    for(;;)
    {
        cout << " New hostinfo " << endl;
        for (j=0; j<total_hosts_count; j++) {
            cout << hosts[j].name << " " << hosts[j].priority << " "
                 << hosts[j].portnum << " " <<hosts[j].id << endl;
            if (assign_id <= hosts[j].id)
                assign_id = hosts[j].id+1;
        }
        cout << "Task server is available...
             " << endl;

cnt = sck -> sev_accept();

if (pipe(pipes) == 0) {
    if ( (childpid = fork()) < 0 )
        fprintf(stderr, "fork error");
    else if (childpid == 0) {
        close(sck ->sockfd);
        cout << "ready to accept data = " << endl;
        // get information about service
        sck->recvint(sck->conct, &key);
        cout << "key = " << key << endl;
        switch(key)
        {
        case FROMTASK:
            break;

        case FROMREG:
            SYN_IO_S(sck->conct);
            sck->recvint(sck->conct, &count);
            SYN_IO_S(sck->conct);
            sck->recvdata(sck->conct, buf, count);
            update_server_list();
            reset();
            break;

        case OSBREQUEST:
            SYN_IO_S(sck->conct);
            sck->recvint(sck->conct, &num);
        
    }
SYN_IO_S(sck->conct);
newid = assign_hosts(num);
msg = count;
sck->sendint(sck->conct, &newid);
SYN_IO_R(sck->conct);
sck->sendint(sck->conct, &count);
SYN_IO_R(sck->conct);
sck->senddata(sck->conct, buf, msg);
reset();
break;

case OSBRETURN:
SYN_IO_S(sck->conct);
sck->recvint(sck->conct, &return_id);
SYN_IO_S(sck->conct);
return_hosts(return_id);
reset();
break;

default:
    break;
}
//close switch
update_mirror();
datasize = write(pipes[1], hosts, BUFSIZ);
exit(0);
} // else if (fork)
close (sck -> conct); // parent process
datasize = read(pipes[0], hosts, BUFSIZ);
cout << "GO BACA AGAIN .......
" << endl << endl;
} //end pipe

} //end for loop

} //end if
else {
    fprintf(stderr, "Can't creat socket..."瘙);
    exit(0);
}

}

void Task::save_info()
{
    ofstream outfile(UPDATEFILE, ios::out);
    if (!outfile) {
        cerr << "cannot open file for output \n";
        exit (-1);
    }

}
outfile << assign_id << endl << endl;
for (int j=0; j<total_hosts_count; j++) {
    outfile << MARK << hosts[j].name << " " << hosts[j].priority << " " << hosts[j].portnum << " " << hosts[j].id << endl;
}
outfile.close();

void Task::unpack_hosts_info()
{
    int i, j, k, index=1, C=0;
    char *id, *priority;

    for (k=0; k<count; k++){
        if (buf[k] == DELM)
            C++;
    }

    for (i=0; i<C; i++){
        unpack(temp[i].arr, index++);
    }
    i = 0;
    j=0;
    while (i<C) {
        strcpy(hosts[j].name, temp[i++].arr);
        hosts[j].priority = atoi(temp[i++].arr);
        strcpy(hosts[j].portnum, temp[i++].arr);
        hosts[j].id = atoi(temp[i++].arr);
        j++;
    }
}

void Task::pack_all_host_info()
{
    char id[5], priority[2];
    int i;
    bzero(id, sizeof(id));
    bzero(priority, sizeof(priority));

    for (i=0; i<total_hosts_count; i++) {
        pack(hosts[i].name);
        itoa(hosts[i].priority, &priority[0]);
        pack(priority);
        pack(hosts[i].portnum);
        itoa(hosts[i].id, &id[0]);
        pack(id);
    }
}
void Task::host_info_from_mirror()
{
    int ok = 1, key, msk, size;
    int total = total_hosts_count;
    SOCKET *mir;

    key = TASKINIT;

    mir = new SOCKET(mirror_port, t_mirror, SOCK_STREAM);
    if(mir->cln_callsock() == 0) {
        fprintf(stderr, "Mirror Died !!!");
    } else {
        msk = mir->sockfd;
        mir->sendint(msk, &key);
        mir->recvint(msk, &ok);
        mir->recvint(msk, &count);
        mir->sendint(msk, &ok);
        mir->recvdata(msk, buf, count);
    }
    unpack_hosts_info();
    reset();
}

void Task::update_mirror()
{
    int ok = 1, key, msk, size;
    int total = total_hosts_count;
    SOCKET *mir;

    pack_all_host_info();
    size = count;
    key = FROMTASK;

    mir = new SOCKET(mirror_port, t_mirror, SOCK_STREAM);
    if(mir->cln_callsock() == 0) {
        fprintf(stderr, "Mirror Died !!!");
    } else {
        msk = mir->sockfd;
        mir->sendint(msk, &key);
        mir->recvint(msk, &ok);
        mir->sendint(msk, &size);
        mir->recvint(msk, &ok);
        mir->senddata(msk, buf, count);
    }
}

void Task::read_newinfo()
{
char ch;

file_fd.open(UPDATEFILE, ios::in);

file_fd >> assign_id;

int i=0;
while (file_fd.get(ch))
{
    if (ch == MARK) {
        file_fd >> hosts[i].name;
        file_fd >> hosts[i].priority;
        file_fd >> hosts[i].portnum;
        file_fd >> hosts[i].id;
        i++;
    }
    if(ch == EOF) break;
    while((file_fd.get(ch)) && (ch != '\n'));
}

file_fd.close();

total_hosts_count = i;
}

int Task::assign_hosts(int num)
{
    int i, j, k, p, q, len;
    i=0; j=0; p=0; q=0;
    while (i<num) {
        if ((hosts[j].priority == 0) && (j<total_hosts_count)){
            hosts[j].id = assign_id;
            hosts[j].p_backup = hosts[j].priority;
            hosts[j].priority = 3;
            pack(hosts[j].name);
            pack(hosts[j].portnum);
            i++;
        }
        else {
            if (j>=total_hosts_count && p<total_hosts_count){
                if (hosts[p].priority == 1 ){
                    hosts[p].id = assign_id;
                    hosts[p].p_backup = hosts[p].priority;
                    hosts[p].priority = 3;
                    pack(hosts[p].name);
                    pack(hosts[p].portnum);
                    i++;
                }
                p++;
            }
        }
    }
else if (p>=totkl_hosts_count && q<total_hosts_count ){
    if (hosts[q].priority == 2 ){
        hosts[q].id = assign_id;
        hosts[q].p_backup = hosts[q].priority;
        hosts[q].priority = 3;
        pack(hosts[q].name);
        pack(hosts[q].portnum);
        i++;
    }
    q++;
} // else
j++;
if (q>=total_hosts_count) {
    num = i;
} //while
}
for(i=0; i<count; i++) {
    cout << buf[i];
}
return assign_id;

void Task::update_server_list()
{
    int k, j, index = 1, hostcount = 0;
    char tmp[1];
    for(k=0; k<count; k++) {
        if ( buf[k] == DELM) {
            hostcount++;
        }
    }
    hostcount = (hostcount/2);
    for (k=0; k < hostcount; k++) {
        unpack(avhosts[k].name, index++);
        unpack(tmp, index++);
        avhosts[k].priority = atoi(tmp);
    }
    /*
    for (k=0; k < total_hosts_count; k++) {

cout << "avhosts => " « avhosts[k].name « " "«
avhosts[k].priority << endl;
}

for (k=0; k < total_hosts_count; k++) {
    if ( hosts[k].priority != 1) {
        hosts[k].priority = 2;
    }
}
*

for (k=0; k<hostcount; k++)
{
    for (j=0; j<total_hosts_count; j++) {
        if (strcmp(hosts[j].name, avhosts[k].name) == 0) {
            if (hosts[j].priority != 3) {
                hosts[j].priority = avhosts[k].priority;
            }
            else {if (hosts[j].priority == 3) {
                hosts[j].p_backup = avhosts[k].priority;
            }
            break;
        }
    }
}

cout << " Host info after update " << endl;
for (j=0; j<total_hosts_count; j++) {
}
//save_info();
}

void Task::return_hosts(int id)
{
    int j;

    for (j=0; j<total_hosts_count; j++) {
        if (hosts[j].id == id) {
            hosts[j].priority = hosts[j].p_backup;
            hosts[j].id = 0;
        }
    }
}

void Task::get_hostinfo()
{
    int i, j;
    char ch;

    file_fd.open(HOSTFILE, ios::in);

    cout << " avhosts => " « avhosts[k].name « " " «
avhosts[k].priority << endl;
}

for (k=0; k < total_hosts_count; k++) {
    if ( hosts[k].priority != 1) {
        hosts[k].priority = 2;
    }
}
*/

for (k=0; k<hostcount; k++)
{
    for (j=0; j<total_hosts_count; j++) {
        if (strcmp(hosts[j].name, avhosts[k].name) == 0) {
            if (hosts[j].priority != 3) {
                hosts[j].priority = avhosts[k].priority;
            }
            else {if (hosts[j].priority == 3) {
                hosts[j].p_backup = avhosts[k].priority;
            }
            break;
        }
    }
}

cout << " Host info after update " << endl;
for (j=0; j<total_hosts_count; j++) {
}
//save_info();
}

void Task::return_hosts(int id)
{
    int j;

    for (j=0; j<total_hosts_count; j++) {
        if (hosts[j].id == id) {
            hosts[j].priority = hosts[j].p_backup;
            hosts[j].id = 0;
        }
    }
}

void Task::get_hostinfo()
{
    int i, j;
    char ch;

    file_fd.open(HOSTFILE, ios::in);
i=0;
while (file_fd.get(ch))
{
    if (ch == MARK) {
        file_fd >> hosts[i].name;
        file_fd >> hosts[i].portnum;
        hosts[i].priority = 4;
        i++;
    }
    if(ch == EOF) break;
    while((file_fd.get(ch)) && (ch != '\n'));
}
file_fd.close();

total_hosts_count = i;
host_info_from_mirror();

}

void Task::delay()
{
    for (int i=0; i<5000000; i++);
}

void Task::reset()
{
    count = 0;
    bzero(buf, sizeof(buf));
}

//******************************************************************************
// Task.C
// Task manager for SPIDER II
//******************************************************************************

#include "task_obj.C"

void main()
{
    int x, myport;
    Task *tk;

    sendflag(T_Flag);
    myport = T_manager_port;
    tk = new Task();

    tk -> get_hostinfo();
    // tk -> host_info_from_mirror();
tk -> waiting();

/******************************************************************************
* t_mirror.C
* Mirror of Task Manager
* SPIDER II
* ******************************************************************************/

#include"task_obj.C"

class T_mirror: private Task
{
    public:
        T_mirror();
        ~T_mirror();
        void waiting();
};

T_mirror::T_mirror()
{
    int i;
    for(i=0;i< MAXHOST;i++)
    {
        bzero(hosts[i].name, sizeof(hosts[i].name));
        bzero(hosts[i].portnum, sizeof(hosts[i].portnum));
        hosts[i].priority = 0;
        hosts[i].id = 0;
    } /* initial HOST */
    for (i=0; i < MAXHOST; i++)
        bzero(avhosts[i].name, sizeof(avhosts[i].name));

    assign_id = 0;
    total_hosts_count = 0;
    count =0;
    myport = T_mirror_port;
    mirror_port = T_manager_port;
    t_mirror = Task_Server;
    bzero(local_host, sizeof(local_host));
    bzero(remote_host, sizeof(remote_host));
    gethostname(local_host, MAXSTR);
    printf("my hostname is %s\n", local_host);
    cout << "myport = " << myport << endl;
    get_hostinfo();
}

T_mirror::~T_mirror()
{ }
void T_mirror::waiting()
{
    int i, j, msg, num, datasize;
    pid_t childpid;
    int cnt, newid, return_id, pipes[2];
    int ok = 1;
    int key = 1;

    if(myport == 0) {
        fprintf(stderr, "Can't get T_manager port");
        exit(0);
    }

    sck = new SOCKET(myport, SOCK_STREAM);

    if (sck ->sev_callsock() >=0)
    {
        for(;;)
        {
            cout << " New hostinfo " << endl;
            for (j=0; j<total_hosts_count; j++) {
                cout << hosts[j].name << " " << hosts[j].priority << " "
                     << hosts[j].portnum << " " <<hosts[j].id << endl;
                if (assign_id <= hosts[j].id)
                    assign_id = hosts[j].id+1;
            }
            cout << "Task Mirror is avaliable..." << endl;

            cnt = sck -> sev_accept();

            if (pipe(pipes) == 0) {
                if ((childpid = fork()) < 0 )
                    fprintf(stderr, "fork error");
                else if (childpid == 0) {
                    close(sck ->sockfd);
                    cout << "ready to accept data = " << endl;
                    // get information about service
                    sck->recvint(sck->conct, &key);
                    cout << "key = " << key << endl;
                    switch(key)
                    {
                        case FROMTASK:
                            SYN_IO_S(sck->conct);
                            sck->recvint(sck->conct, &count);
                            SYN_IO_S(sck->conct);
                            sck->recvdata(sck->conct, buf, count);
                            unpack_hosts_info();
                            break;
                        case TASKINIT:
                            SYN_IO_S(sck->conct);
                            pack_all_host_info();
                    }
                }
            }
        }
    }
}
msg = count;
sck->sendint(sck->conct, &count);
SYN_IO_R(sck->conct);
sck->senddata(sck->conct, buf, msg);
reset();
break;

case FROMREG:
    SYN_IO_S(sck->conct);
sck->recvint(sck->conct, &count);
SYN_IO_S(sck->conct);
sck->recvdata(sck->conct, buf, count);
update_server_list();
reset();
break;

case OSBREQUEST:
    SYN_IO_S(sck->conct);
sck->recvint(sck->conct, &num);
cout << "num in OSBREQUEST= " << num << endl;
SYN_IO_S(sck->conct);
newid = assign_hosts(num);
msg = count;
cout << "count = " << count;
sck->sendint(sck->conct, &newid);
SYN_IO_R(sck->conct);
sck->sendint(sck->conct, &count);
SYN_IO_R(sck->conct);
sck->senddata(sck->conct, buf, msg);
reset();
break;

case OSBRETURN:
    SYN_IO_S(sck->conct);
sck->recvint(sck->conct, &return_id);
SYN_IO_S(sck->conct);
return_hosts(return_id);
reset();
break;

default:
    break;

} //close switch

data_size = write(pipes[1], hosts, BUFSIZ);
exit(0);
} //else if (fork)
close(sck -> conct); // parent process
data_size = read(pipes[0], hosts, BUFSIZ);
cout << "GO BACA AGAIN ......." << endl << endl;
} //end pipe

} //for loop
```cpp
// 
else {
  fprintf(stderr, "Can't creat socket...");
  exit(0);
}

void main()
{
  int x, myport;
  T_mirror *tm;
  tm = new T_mirror();
  tm -> waiting();
}

// ***********************************************
// *** reg.h ***
// *** Header file for registry server ***
// ***********************************************

#include <iostream.h>
#include <fstream.h>
#include <stdio.h>
#include <string.h>
#include "Socket.C"
#include "profile.h"
#include "util.C"

#define SYN_IO_S(x) (sock->sendint(x, &ok))
#define SYN_IO_R(x) (sock->recvint(x, &ok))
#define INT sizeof(int)
#define STRING (sizeof(char)*MAXSTR)

class AV_HOST
{
  public:
    char name[MAXSTR];
    int priority;
};

class HOST
{
  public:
    char name[MAXSTR];
};

class Registry
{
public:
    Registry();
    ~Registry();
    void hostsent();

private:
    AV_HOST avhosts[50];
    HOST hosts[50];
    char buf[MAXBUFSIZE];
    char local_host[MAXSTR];
    char remote_host[MAXSTR];

    int myport, firstport;
    int count, key, num;
    int total_hosts_count;
    int get_server_list();
    SOCKET *sock;
    fstream file_fd;
    void start_test();
    void unpack(char *pk, int off);
    void pack(char *pk);
    // void reset();


************
// ***
// *** registry server for SPIDER II
// ***

#include "reg.h"

Registry::Registry()
{
    int i;

    for (i=0; i<50; i++) {
        bzero(avhosts[i].name, sizeof(avhosts[i].name));
        bzero(hosts[i].name, sizeof(hosts[i].name));
        hosts[i].port = 0;
        hosts[i].priority = 4;
    }

    total_hosts_count = 0;
    count = 0;
    firstport = Reg_port;
    // bzero(service, sizeof(service));
    bzero(local_host, sizeof(local_host));
    bzero(remote_host, sizeof(remote_host));
    gethostname(local_host, MAXSTR);
    printf("my hostname is %s\n", local_host);
Registry::~Registry()
{
    delete sock;
}

int Registry::get_server_list()
{
    int i;
    char ch;
    char temp[MAXSTR];
    bzero(temp, STRING);
    file_fd.open(DATA_FILE, ios::in);
    for(i=0; i < 50; i++) { // re-initialize the hosts array
        bzero(hosts[i].name, sizeof(hosts[i].name));
    }
    if(!file_fd) {
        fprintf(stderr, "table file open error!!");
        return 0;
    }
    i = 0; // record how many hosts got
    while(file_fd.get(ch)) {
        if(ch == MARK) {
            hosts[i].port=firstport+i;
            file_fd >> hosts[i++].name;
        }
        if(ch == EOF) break;
        while((file_fd.get(ch)) && (ch != '\n'));
    }
    file_fd.close();
    // cout << "hosts = " << i << endl;
    return i;
}

void Registry::start_hosts()
{
    int len, i, j, cnt;
    char cmd[256];
    char *R=RSH;
    char *Path=Spiderpath;
    char *service="reg_test";
    char aport[4];
    pid_t childpid;
    for (int i=0; i<total_hosts_count; i++){
        if ((childpid = fork()) < 0) {
fprintf(stderr, "fork error");
_exit(0);
}
if (childpid == 0) {
    bzero(cmd, sizeof(cmd));
    cnt = 0;
    len = strlen(R);
    for (j=0; j < len; j++)
        cmd[cnt++] = R[j];
    cmd[cnt++] = ' '; 
    len = strlen(hosts[i].name);
    for (j=0; j < len; j++)
        cmd[cnt++] = hosts[i].name[j];
    cmd[cnt++] = ' '; 
    len = strlen(Path);
    for (j=0; j < len; j++)
        cmd[cnt++] = Path[j];
    cmd[cnt++] = '/'; 
    len = strlen(service);
    for (j=0; j < len; j++)
        cmd[cnt++] = service[j];
    cmd[cnt++] = ' '; 
    itoa(hosts[i].port, &aport[0]);
    len = 4;
    for (j=0; j < len; j++)
        cmd[cnt++] = aport[j];
    // cmd[cnt++] = ' ';
    system(cmd);
    exit(0);
}
} // end loop

sleep(8);

void Registry::broadcasting()
{
    int sk, ok = 1, i, size, name_size, index;
    pid_t child_pid, pid;
    SOCKET *sck[total_hosts_count];
    char name[10], cpuidle[3];
    int pipes[2];

    for (i=0; i < total_hosts_count; i++) {
        sck[i] = new SOCKET(hosts[i].port, hosts[i].name, SOCK_STREAM);
    }

    for (i=0; i < total_hosts_count; i++){
        if (pipe(pipes)==0) {
            if ((child_pid = fork()) < 0) {
                fprintf(stderr, "fork error....");
            }
        }
    }
}
_exit(0);
}

if (child_pid == 0) {
  if (sck[i] -> cln_callsock() == 0) {
    // fprintf(stderr, "Socket does not working");
    strcpy(name, "none|0|");
    size = 7;
    name_size = write(pipes[1], name, size);
    _exit(0);
  }

  if (child_pid == 0) {
    sk = sck[i] -> sockfd;
    sck[i] -> sendint(sk, &key);
    sck[i] -> recvint(sk, &size);
    sck[i] -> sendint(sk, &ok);
    sck[i] -> recvdata(sk, name, size);
    sck[i] -> sendint(sk, &ok);
    name_size = write(pipes[1], name, size);
  }

  close(sck[i] -> conct);
  _exit(0);
}

name_size = read(pipes[0], avhosts[i].name, BUFSIZ);
} // end pipe

index = i;
bzero(cpuidle, sizeof(cpuidle));
strcpy(buf, avhosts[i].name);
count = sizeof(buf);
bzero(avhosts[i].name, sizeof(avhosts[i].name));
unpack(avhosts[i].name, index++);
unpack(cpuidle, index++);
avhosts[i].idle = atoi(cpuidle);
cout << "Avhosts " << i << " => " << avhosts[i].name
    << " " << avhosts[i].idle << endl;
}

//for loop

void Registry::unpack(char *out, int offset) {
  char ch;
  int i = 0, j = 1;
  int k = 0;

  if (j == offset) {
    while (buf[i] != DELM) {
      out[k++] = buf[i++];
    }
  } else {
    while (i < count) {
      while (buf[i++] != DELM);
  }

87
j++;  
  if (j == offset) {  
    while (buf[i] != DELM) {  
      out[k++] = buf[i++];  
      i = count;  
    }  
  }  
}

void Registry::pack(char *pk)
{
  int i = 0, len = strlen(pk);
  for (i = 0; i < len; i++)
    buf[count++] = pk[i];
  buf[count++] = DELM;
}

void Registry::new_hosts_list()
{
  for (int i = 0; i < total_hosts_count; i++) {
    if (strcmp(avhosts[i].name, "none") != 0) {
      for (int j = 0; j < total_hosts_count; j++) {
        if (strcmp(avhosts[i].name, hosts[j].name) == 0) {
          if (avhosts[i].idle > 50)
            hosts[j].priority = 0;
          else if (avhosts[i].idle > 30)
            hosts[j].priority = 1;
          else if (avhosts[i].idle > 0)
            hosts[j].priority = 2;
        }
      }
    }
  }
}

void Registry::hostsent()
{
  int j, h, size;
  int sk, ok = 1;
  int port = T_manager_port;
  int tflag = T_Flag;
  char temp[1];
  bzero(temp, 1);
  j = get_server_list();
  total_hosts_count = j;
  start_hosts();
  broadcasting();
  new_hosts_list();
for (int i=0; i<total_hosts_count; i++) {
    cout << hosts[i].name << " " << hosts[i].priority << endl;
}
count =0;
bzero(buf, sizeof(buf));

for (h=0; h<j; h++) {
    pack(hosts[h].name);
    itoa(hosts[h].priority, &temp[0]);
    pack(temp);
}

size = count;
key = PROMREG;

sock = new SOCKET(port, Task_Server, SOCK_STREAM);

if(sock -> cln_callsock() == 0) {
    fprintf(stderr, "Task is not working");
    tflag = M_Flag;
    sock = new SOCKET(T_mirror_port, Task_Mirror, SOCK_STREAM);
    if(sock -> cln_callsock() == 0) {
        fprintf(stderr, "Mirror is not working");
        tflag = 2;
        exit(0);
    }
}

sendflag(tflag);
sk = sock -> sockfd;
sock->sendint(sk, &key);
SYN_IO_R(sk);
sock->sendint(sk, &count);
SYN_IO_R(sk);
sock->senddata(sk, buf, size);
cout << " Update to Task Server ... " << endl;
}

void main()
{
    Registry *reg;

    reg = new Registry;
    reg -> hostsentO;
    exit(0);
}
#include <iostream.h>
#include <fstream.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#include "Socket.C"
#include "cpuusage.C"
#include "util.C"

#define SYN_IO_R(x) (sck->recvint(x, &ok))
#define SYN_IO_S(x) (sck->sendint(x, &ok))

class Server
{
public:
    Server();
    void waiting(int port);
    void print();
    int perms, fd;

private:
    int myport, sk, count;
    SOCKET *sck;
    CPUusage *cusage;
    char name[20], buf[20];

    void pack(char *pk);
    void unpack(char *out, int offset);

};

Server::Server()
{
    bzero(name, sizeof(name));
    bzero(buf, sizeof(buf));
    cusage = new CPUusage;
}
void Server::pack(char *pk)
{
    int i=0, len = strlen(pk);
    for(i=0;i<len;i++)
        buf[count++] = pk[i];
    buf[count++] = DELM;
}

void Server::unpack(char *out, int offset)
{
    char ch;
    int i=0, j=1,
    int k=0;
    if( j == offset) {
        while(buf[i] != DELM) {
            out[k++] = buf[i++];
        }
    }
    else {
        while(i < count) {
            while(buf[i++] != DELM);
            j++;
            if(j == offset) {
                while(buf[i] != DELM) {
                    out[k++] = buf[i++];
                }
                i = count;
            }
        }
    }
}

void Server::waiting(int port)
{
    int datasize, sbuf;
    int i, j, num=40;
    pid_t childpid,pid;
    int cnt, key, x;
    int ok = 1;
    char cpuidle[3];
    myport = port;
    count =0;
    if(myport == 0) {
        fprintf(stderr, "Can't get local port");
        exit(0);
    }
    getname(&name[0]);
cusage->get_cpu_idle(cpuidle);
pack(name);
pack(cpuidle);
datasize = strlen(buf);
sbuf = strlen(buf);

sck = new SOCKET(myport, SOCK_STREAM);

if (sck->sev_callsock() >= 0)
{
    cnt = sck -> sev_accept();
    close(sck -> sockfd);
    sck -> recvint (sck->conct, &key);
    sck -> sendint (sck->conct, &datasize);
    sck -> recvint (sck->conct, &ok);
    sck -> senddata (sck->conct, buf, sbuf);
    sck -> recvint (sck->conct, &ok);
    close (sck -> conct);
} else {
    fprintf(stderr,"Can creat socket...");
    exit(0);
}

main(int argc, char *argv[])
{
    port;
    Server *svr;

    if (argc <=1) {
        printf(stderr," Port not found ");
        exit(0);
    } else {
        port = atoi(argv[1]);
        svr = new Server;
        svr -> waiting(port);
        exit(0);
    }
}

// **********************
// *** Registry Daemon ***
// **********************

#include <iostream.h>
#include <stdio.h>
```c
#include <stdlib.h>

void main() {
    for (;;) {
        system("reg ");
        sleep(240);
        system("killall reg");
    }
}

// Registry Mirror Daemon

#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>

void main() {
    sleep(120);
    for (;;) {
        system("reg ");
        sleep(240);
        system("killall reg");
    }
}

// Testing of CPU Idle

#include <stdio.h>
#include <iostream.h>
#include <stdlib.h>
#include <fstream.h>
#include <string.h>

class CPUINFO {
    char info[6];
};

class CPUusage {
    public:
        CPUusage();
};
```
void cpu_info();
int get_cpu_idle();
void get_cpu_idle(char *id);
void print();

private:
    fstream file_fd;
    CPUINFO cinfo[40];
    int count;
};

CPUusage::CPUusage()
{
    char ch;
    system("vmstat > /tmp/CPUstat");
    file_fd.open("/tmp/CPUstat", ios::in);
    count = 0;

    while (file_fd.get(ch)) {
        file_fd >> cinfo[count].info;
        if (ch == EOF) break;
        count++;
    }

    file_fd.close();
    system("rm -r /tmp/CPUstat");
}

CPUusage::~CPUusage()
{
}

int CPUusage::get_cpu_idle()
{
    int id = atoi(cinfo[count-2].info);
    return id;
}

void CPUusage::get_cpu_idle(char *id)
{
    strcpy(id, cinfo[count-2].info);
}

void CPUusage::print()
{
    cout << "idle " << cinfo[count-2].info << endl;
    cout << "sys time " << cinfo[count-3].info << endl;
    cout << "usertime " << cinfo[count-4].info << endl;
}
cout << endl;
}

// ************ osb.h ************
// Header file for Object Service broker
// ************ SPIDER II 2000 ************
#include <iostream.h>
#include <fstream.h>
#include <stdio.h>
#include <string.h>
#include "Socket.C"
#include "profile.h"
#include "util.C"
#define SYN_IO_S(x) (sock->sendint(x, &ok))
#define SYN(IO_R(x) (sock->recvint(x, &ok))
#define INT sizeof(int)
#define STRING (sizeof(char)*MAXSTR)
#define MAXHOST 100

class TEMP
{
    public:
        char arr[MAXSTR];
    private:
        HOST hosts[MAXHOST];
        TEMP temp[MAXHOST];
        char buf[MAXBUFSIZE];
        char myname[MAXBUFSIZE];
        int sk, id, num;
        int count, T_port, myport;
        int total_hosts_count;
        char T_server[10];
        SOCKET *sock;
        char *cmd[30];
        void unpack(char *pk, int off);
    public:
        OSB();
}
void spider_spawn(char *service, int num);
void start_server(char *service);
void send_request(int num);
void send_data(int *data, int sz);
void send_data_matrixp(int *data1, int *data2, int datasize);
int got_list();
int get_return(int *data, int sz);
int get_return(int *data);
int get_return_matrixp(int mdata[9][MAX]);

// Object Service Broker for SPIDER II

#include "osb.h"

OSB::OSB()
{
    int i, f;

    bzero(buf,sizeof(buf));
    for(i=0;i< MAXHOST;i++)
    {
        bzero(hosts[i].name, sizeof(hosts[i].name));
        bzero(hosts[i].port, sizeof(hosts[i].port));
        // hosts[i].port=0;
    }
    /* initial HOST */
    total_hosts_count = 0;

    gethostname(mname, MAXBUF_SIZE);

    f = getflag();

    if (f==0) {
        T_port = T_manager_port;
        strcpy(T_server, Task_Server);
    } else if (f==1) {
        T_port = T_mirror_port;
        strcpy(T_server, Task_Mirror);
    }
}

OSB::~OSB()
{
    delete sock;
}
void OSB::unpack(char *out, int offset)
{
    char ch;
    int i=0, j=1;
    int k=0;

    if (j == offset) {
        while(buf[i] != DELM) {
            out[k++] = buf[i++];
        }
    }
    else {
        while(i < count) {
            while(buf[i++] != DELM);
            j++;
            if (j == offset) {
                while(buf[i] != DELM) {
                    out[k++] = buf[i++];
                }
                i = count;
            }
        }
    }
}

int OSB::got_list()
{
    int index=1, i;
    int C=0;

    for (i=0; i<count; i++){
        if (buf[i] == DELM)
            C++;
    }

    for (i=0; i<C; i++){
        unpack(temp[i].arr, index++);
    }
    i = 0;
    int j=0;
    while (i<C) {
        strcpy(hosts[j].name, temp[i++].arr);
        strcpy(hosts[j].port, temp[i++].arr);
        j++;
    }

    cout << "Got hosts : " << endl;
    for (int i=0; i<j; i++){
        cout << hosts[i].name << " " << hosts[i].port << endl;
    }
}
return j;
}

void OSB::send_request(int num)
{
    int ok = 1;
    int re;
    re = OSBREQUEST;
    sock = new SOCKET(T_port, T_server, SOCK_STREAM);
    if(sock -> cln_callsock() == 0) {
        fprintf(stderr, "Task Manager does not working\n");
        T_port = T_mirror_port;
        strcpy(T_server, Task__Mirror);
        sendflag(M_Flag);
        sock = new SOCKET(T_port, T_server, SOCK_STREAM);
        if(sock -> cln_callsock() == 0) {
            fprintf(stderr, "Task Mirror does not working either\n");
            exit(0);
        }
    }
    sk = sock -> sockfd;
    sock->sendint(sk, &re);
    SYN_IO_R(sk);
    sock->sendint(sk, &num);
    SYN_IO_R(sk);
    sock->recvint(sk, &id);
    SYN_IO_S(sk);
    sock->recvint(sk, &count);
    SYN_IO_S(sk);
    sock->recvdata(sk, buf, count);
    cout « "ID = " « id « endl;
    close(sock -> sockfd);
}

void OSB::spider_spawn(char ^service, int num_need)
{
    send_request(num_need);
    num = got_list();
    cout « "Got hosts : " « endl;
    for (int i=0; i<num; i++){
        cout « "host= " « hosts[i].name « " " « hosts[i].port « endl;
    }
    start_server(service);
}
void OSB::start_server(char *service)
{

    int len, i, j, cnt;
    char cmd[256];
    char *R=RSH;
    char *Path=Spiderpath;
    pid_t childpid;

    cout << "got_num= " << num << endl;
    for (int i=0; i<num; i++) {

        if ((childpid = fork()) < 0) {
            fprintf(stderr, "fork error");
            _exit(0);
        }

        if (childpid == 0) {
            bzero(cmd, sizeof(cmd));
            cnt = 0;

            len = strlen(R);
            for (j=0; j<len; j++)
                cmd[cnt++]=R[j];
            cmd[cnt++]=';';
            len = strlen(hosts[i].name);
            for (j=0; j<len; j++)
                cmd[cnt++]=hosts[i].name[j];
            cmd[cnt++]=';';
            len = strlen(Path);
            for (j=0; j<len; j++)
                cmd[cnt++]=Path[j];
            cmd[cnt++]='/';
            len = strlen(service);
            for (j=0; j<len; j++)
                cmd[cnt++]=service[j];
            cmd[cnt++]=';';
            len = strlen(hosts[i].port);
            for (j=0; j<len; j++)
                cmd[cnt++]=hosts[i].port[j];
            cmd[cnt++]=';';

            cout << "Start Object-Server: " << hosts[i].name << endl;
            system(cmd);
            exit(0);
        }
    } // end loop
    if (num <= 8 )
        sleep(3);
    else sleep(5);
void OSB::send_data_matrixp(int *datal, int *data2, int datasize)
{
    pid_t child_pid, pid;
    int num=9;
    int i, m, len, portnum, ok=1;
    SOCKET *sck[num];
    int *arr1[num], *arr2[num];
    int arrsize1 = datasize/num;
    int arrsize2 = datasize/(num/3);
    int sizel, size2, count1, count2, sub_group;

    myport = atoi(hosts[0].port)+1000;

    for (i=0; i<num; i++) {
        arrl[i] = &datal[i*arrsizel];
        m = (i%3);
        arr2[i] = &data2[m*arrsize2];
    }

    for (i=0; i< num; i++){
        sck[i] = new SOCKET(atoi(hosts[i].port), hosts[i].name,
            SOCK_STREAM);
    }

    i=0;
    for (i=0; i< num; i++){
        if ((child_pid = fork()) < 0){
            fprintf(stderr, "fork error....");
                _exit(0);
        }

        if (child_pid == 0){
            if (sck[i] -> cln_callsock() == 0) {
                fprintf(stderr, "Socket does not working");
                exit(0);
            } else {
                sub_group = i;
                len=strlen(mynname);
                portnum = myport;
                count1 = sizel = arrsize1*4 ;
                count2 = size2 = arrsize2*4 ;
                sk = sck[i] -> sockfd;

                sck[i]->sendint(sk, &len);
                sck[i]->recvint(sk, &ok);
                sck[i]->senddata(sk, myname, strlen(mynname));
                sck[i]->recvint(sk, &ok);
                sck[i]->sendint(sk, &portnum);
                sck[i]->recvint(sk, &ok);
        }
    }
sck[i]->sendint(sk, &size1);
sck[i]->recvint(sk,&ok);
sck[i]->senddata(sk, arr1[i], count1);
sck[i]->recvint(sk, &ok);
sck[i]->sendint(sk, &size2);
sck[i]->recvint(sk, &ok);
sck[i]->senddata(sk, arr2[i], count2);
sck[i]->recvint(sk, &ok);
sck[i]->sendint(sk, &sub_group);
} // else
    close(sk);
  _exit(0);
} // if (child_pid)

} // for loop

void OSB::send_data(int *data, int sz)
{
    int i, size, ok=1, len, portnum, x;
    SOCKET *sck[num];
    int chrsize, count;
    int *arr[num], *tmp;
    pid_t child_pid, pid;
    int bs = sz/num;
    myport = atoi(hosts[0].port)+1000;

    for (i=0; i<num; i++) {
        arr[i] = &data[i*bs];
    }

    for (i=0; i< num; i++)
    {
        sck[i] = new SOCKET(atoi(hosts[i].port), hosts[i].name,
                         SOCK_STREAM);
    }

    i=0;
    for (i=0; i< num; i++){

        if ((child_pid = fork()) < 0){
            fprintf(stderr, "fork error....");
            _exit(0);
        }

        if (child_pid == 0){

            if(sck[i] -> cln_callsock() == 0) {
                fprintf(stderr, "Socket does not working");
                exit(0);
            }

            else {
                len=strlen(mynname);
                portnum = myport;

count = size = bs*4.;
sk = sck[i] -> sockfd;

sck[i] -> sendint(sk, &len);
sck[i] -> recvint(sk, &ok);
sck[i] -> senddata(sk, myname, strlen(myname));
sck[i] -> recvint(sk, &ok);
sck[i] -> sendint(sk, &portnum);
sck[i] -> recvint(sk, &ok);
sck[i] -> sendint(sk, &size);
sck[i] -> recvint(sk, &ok);
sck[i] -> senddata(sk, arr[i], count);

close(sk);
_exit(0);

int OSB::get_return(int *data, int sz)
{
    int i, j=0, total_size=0;
    pid_t childpid, pid;
    int cnt, size, x, datasize;
    int ok = 1, z=0;
    SOCKET *osbsck;
    int bs = sz/num;
    int *arr, *arr1, pipes[2];

    osbsck = new SOCKET(myport, SOCK_STREAM);
    if (osbsck->sev_callsock() >= 0)
    {
        for (j=0; j < num; j++)
        {
            arr1 = &data[j*bs];
            cnt = osbsck -> sev_accept();
            if (pipe(pipes)==0) {
                if ((childpid = fork()) < 0 )
                    fprintf(stderr, "fork error");
                if (childpid == 0) {
                    close(osbsck -> sockfd);
                    osbsck -> recvint(osbsck->conct, &size);
                    osbsck -> sendint(osbsck->conct, &ok);
                    osbsck -> recvdata(osbsck->conct, arr1, size);
                    datasize = write(pipes[1], arr1, size);
                    cout << "Size in fork = " << size << endl;
                    close(osbsck -> conct);
                    _exit(0); // end fork
                }
            }
        }
    }
}


```c++
do {
    arr=&arrl[size];
    datasize = read(pipes[0], arr, BUFSIZ);
    size = size + datasize/4;
} while (datasize >= 4096);

} // end pipe
} // end loop

else {
    fprintf(stderr,"Can creat socket...");
    exit(0);
    return 0;
}

cout << "Size in perent " << size << endl;
cout << "num in pernet = " << num << endl;
return total_size;

}

int OSB::get_return_matrixp(int mdata[9][MAX])
{

    int i, j=0, total_size=0;
    pid_t childpid, pid;
    int cnt, size, x, datasize ;
    int ok = 1, z=0;
    SOCKET *osbsck;
    int *arr, *arrl, pipes[2];

cout << "myport in get_return = " << myport << endl;
    osbsck = new SOCKET(myport, SOCK_STREAM);

    if (osbsck->sev_callsock() >= 0)
    {
        for (j=0; j < num; j++)
        {
            size = 0;

            // arrl = &mdata[total_size];
            arrl = &mdata[j][0];

            cnt = osbsck -> sev_accept();

            if (pipe(pipes)==0) {
                if ((childpid = fork()) < 0 )
                    fprintf(stderr, "fork error");
                if (childpid == 0) {
                    close(osbsck -> sockfd);
                    osbsck -> recvint (osbsck->conct, &size);
```
osbsck -> sendint(osbsck->conct, &ok);
osbsck -> recvdata (osbsck-> conct, arrl, size);
write(pipes[1], arrl, size);
close (osbsck -> conct);
// close (pipes[1]);
_exit(0);
} // end fork

else {
do {
arr=&arrl[size];
datasize = read(pipes[0], arr, BUFSIZ); size = size + datasize/4;
} while (datasize >= 4000);
cout << " Got from station " << j << endl;
cout << size << " numbers" << endl;
total_size = total_size + size;
}
// close (pipes[2]);
} // end pipe
} // end loop

else {
fprintf(stderr,"Can creat socket...");
exit(0);
}
return size;
}

int OSB::get_return(int *data)
{
int i, j=0, total_size=0;
pid_t childpid, pid;
int cnt, size, x, datasize;
int ok = 1, z=0;
SOCKET *osbsck;
int bs;
int *arr, *arrl, pipes[2];
osbsck = new SOCKET(myport, SOCK_STREAM);
if (osbsck->sev_callsock() >= 0) {
for (j=0; j < num; j++) {
arrl = &data[total_size];
size = 0;
cnt = osbsck -> sev_accept();
if (pipe(pipes)==0) {
if ((childpid = fork()) < 0 )
    fprintf(stderr, "fork error");
if (childpid == 0) {
    close(osbsck -> sockfd);
    osbsck -> recvint (osbsck->conct, &size);
    osbsck -> sendint(osbsck->conct, &ok);
    osbsck -> recvdata (osbsck-> conct, arr1, size);
    write(pipes[1], arr1, size);
    close (osbsck -> conct);
    // close(pipes[1]);
    _exit(0);
} // end fork

else {
    do {
        arr=&arr1[size];
        datasize = read(pipes[0], arr, BUFSIZ);
        size = size + datasize/4;
    } while (datasize >= 4096);
    cout << "Got from station " << j << endl;
    cout << size << " numbers" << endl;
    total_size = total_size + size;
}
    // close(pipes[2]);
} // end pipe
} // end loop

else {
    fprintf(stderr,"Can creat socket...");
    exit(0);
    return 0;
}
    cout << "Total size got = " << total_size << endl;
    return total_size;
}

// ********** osb_slv.h **********
// Header file for
// Object Service broker -- slave class
// ****************************************

#include <iostream.h>
#include <fstream.h>
#include <stdio.h>
#include <string.h>
#include "Socket.C"
#include "profile.h"

#define SYN_IO_S(x) (sck->sendint(x, &ok))
#define SYN_IO_R(x) (sck->recvint(x, &ok))
#define INT sizeof(int)
#define STRING (sizeof(char)*MAXSTR)
#define MAXHOST 100

class osb_slv
{
public:
    osb_slv(int port);
    ~osb_slv();
    int get_data(int *arr);
    int get_data_matrixp(int *arr1, int *arr2);
    void return_data(int *arr, int size);
    void return_data_matrixp(int *arr, int array_size);
    char osbname[MAXBUFSIZE];
    int osbport;

private:
    int sk, id, num;
    int count, size;
    int total_hosts_count;
    int port, matrix_group;
    char myname[MAXBUFSIZE];
};

#include "osb_slv.h"

osb_slv::osb_slv(int myport)
{
    port = myport;
    size = 0;
    bzero(osbname, MAXBUFSIZE);
    osbport = 0;
    getname(myname);
}

osb_slv::~osb_slv()
{
    exit(0);
}

int osb_slv::get_data_matrixp(int *arr_a, int *arr_b)
{
    SOCKET *sck;
int ok=1, osbport=0;
int cnt, array_size_a, len = 0;
int size1, size2;

if(port == 0) {
    fprintf(stderr, "Can't get T_manager port");
    exit(0);
}

sck = new SOCKET(port, SOCK_STREAM);

if (sck ->sev_callsock() >=0) {
    cnt = sck -> sev_accept();
    close(sck -> sockfd);
    sck->recvint(sck->conct, &len);
    sck->sendint(sck->conct, &ok);
    sck->recvdata(sck->conct, osbname, len);
    sck->sendint(sck->conct, &ok);
    sck->recvint(sck->conct, &osbport);
    sck->sendint(sck->conct, &ok);
    sck->recvint(sck->conct, &size1);
    sck->sendint(sck->conct, &ok);
    sck->recvdata(sck->conct, arr_a, size1);
    sck->sendint(sck->conct, &ok);
    sck->recvint(sck->conct, &size2);
    sck->sendint(sck->conct, &ok);
    sck->recvdata(sck->conct, arr_b, size2);
    sck->sendint(sck->conct, &ok);
    sck->recvint(sck->conct, &matrix_group);
    osbport1 = osbport;
    close (sck -> conct);
}
else {
    fprintf(stderr, "Can not creat Socket....");
    exit(0);
}

array_size_a = size1/4;
return array_size_a;
}

int osb_slv::get_data(int *arr) {

    SOCKET *sck;
    int ok=1, osbport=0;
    int cnt, array_size, len = 0;
    
    if(port == 0) {
        fprintf(stderr, "Can't get T_manager port");
        exit(0);
    }

    array_size_a = size1/4;
    return array_size_a;
}
sck = new SOCKET(port, SOCK_STREAM);

if (sck -> sev_callsock() >=0) {
    cnt = sck -> sev_accept();
    close(sck -> sockfd);
    sck -> recvint(sck -> conct, &len);
    sck -> sendint(sck -> conct, &ok);
    sck -> recvdata(sck -> conct, osbname, len);
    sck -> sendint(sck -> conct, &ok);
    sck -> recvint(sck -> conct, &osbport);
    sck -> sendint(sck -> conct, &ok);
    sck -> recvint(sck -> conct, &size);
    sck -> sendint(sck -> conct, &ok);
    sck -> recvdata(sck -> conct, arr, size);
    osbportl = osbport;
    close (sck -> conct);
}
else {
    fprintf(stderr, " Can not creat Socket....");
    exit(0);
}

array_size = size/4;
return array_size;
}

void osb_slv::return_data(int *arr, int array_size) {
    int ok=1, tries =0;
    int cnt, s, si;
    SOCKET *sock;
    s = si = array_size*4;
    sock = new SOCKET(osbport, osbname, SOCK_STREAM);

    trymore: if(sock -> cln_callsock() == 0) {
            if (tries++ < 8) {
                goto trymore;
            }
            else {
                fprintf(stderr, "Socket does not working");
                exit(0);
            };
        }
    }
else {
    sk = sock-> sockfd;
    sock -> sendint(sk, &s);

    osbportl = osbport;
    close (sck -> conct);
}
else {
    fprintf(stderr, " Can not creat Socket....");
    exit(0);
}

return array_size;
}
sock -> recvint(sk,&ok);
sock -> senddata(sk, arr, si);

}  
close(sk);
}

void osb_slv::return_data_matrixp(int *arr, int array_size)
{
int ok=1, tries =0;
int cnt, s, si;
SOCKET *sock;
arr[array_size++]=matrix_group;
si= si= array_size*4;

sock=new SOCKET(osbport, osbname, SOCK_STREAM);
trymore: if(sock -> cln_callsock() == 0) {
    if (tries++ < 8) {
        goto trymore;
    }
else {
    fprintf(stderr, "Socket does not working");
    exit(0);
}
}

else {
    sk = sock-> sockfd;
sock -> sendint(sk, &s);
sock -> recvint(sk,&ok);
sock -> senddata(sk, arr, si);

}
    close(sk);

}

/****************************
***** Socket interface *****
******************************/

#ifndef SOCKET_C
#define SOCKET_C
#include: <stdio.h>
#include <string.h>
#include </usr/include/sys/unistd.h>
#include <fcntl.h>
#include: <stdlib.h>

#endif
```cpp
#include <usr/include/sys/signal.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#define MAXLENGTH 1000
#define MAXHOSTLEN 50

class SOCKET {
    private:
        int type; // protocol type
        char local_hostname[20]; // my name
        sockaddr_in sa, sin; // my socket add. and client's
        hostent *hp;
    public:
        int sockfd;
        int connect;
        int recvdata(int sk, void *msg, int len);
        int senddata(int sk, void *msg, int len);
        int recvint(int sk, iht *msg);
        int sendint(int sk, iht *msg);
        SOCKET(int port);
        SOCKET(int port, char *hostname);
        SOCKET(int port, char *hostname, int protocol);
        SOCKET(int port, int protocol);
        ~SOCKET();
        int sev_callsock();
        int sev_accept();
    
        int cln_callsock();
};

/*
 * SOCKET constructor for server use.
 * using TCP by default.
 * parameter: int port.
 */

SOCKET::SOCKET(int port) {

    hp = new hostent;
    sockfd = 0;
    type = SOCK_STREAM;
    gethostname(local_hostname, MAXHOSTLEN);
    hp = gethostbyname(local_hostname);
    if (hp == NULL) exit(1);
    bzero(&sa, sizeof(sa));
    sa.sin_family = AF_INET;
    sa.sin_addr.s_addr = htonl(INADDR_ANY);
    sa.sin_port = htons((u_short) port);
    
    int sev = socket(AF_INET, SOCK_STREAM, 0);
    bzero(&sa, sizeof(sa));
    sa.sin_family = AF_INET;
    sa.sin_addr.s_addr = htonl(INADDR_ANY);
    sa.sin_port = htons((u_short) port);
?
```
/ *  SOCKET constructor for server use.
  *  parameter: int port, int protocol.
  *  user specifies the protocol he/she wants to use.
  *
*/

SOCKET::SOCKET(int port, int protocol)
{
    hp = new hostent;
    switch(protocol) {
    case SOCK_STREAM:
        type = SOCK_STREAM;
        break;
    case SOCK_DGRAM:
        type = SOCK_DGRAM;
        break;
    case SOCK_RAW:
        type = SOCK_RAW;
        break;
    default:
        type = SOCK_STREAM;
        break;
    }
    sockfd = 0;
    gethostname(local_hostname, MAXHOSTLEN);
    hp = gethostbyname(local_hostname);
    if(hp == NULL) exit(1);
    bzero(&sa, sizeof(sa));
    sa.sin_family = AF_INET;
    sa.sin_addr.s_addr = htonl(INADDR_ANY);
    sockfd = socket(type, AF_INET, 0);
    if(sockfd < 0) {
        perror("socket");
        exit(1);
    }
    sockfd = connect(sockfd, (struct sockaddr *) &sa, sizeof(sa));
    if(sockfd < 0) {
        perror("connect");
        exit(1);
    }
   
} /*
 *  SOCKET constructor for client use
 *  parameter: int port, char *hostname
 * */

SOCKET::SOCKET(int port, char *hostname)
{
    sockfd = 0;
    type = SOCK_STREAM;
    if (!hp) {
        perror("gethostbyname");
    }
    bzero((char *) &sin, sizeof(sin));
    bcopy(hp->h_addr, (char *) &sin.sin_addr, hp->h_length);
    sin.sin_family = AF_INET;
    sin.sin_port = htons((u_short) port);
}

/ *
* SOCKET constructor for client use
* parameter: int port, char *hostname, int protocol
* user can specifies the protocol he/she wants to use
*/
SOCKET::SOCKET(int port, char *hostname, int protocol)
{
    sockfd = 0;
    switch(protocol) {
    case SOCK_STREAM:
        type = SOCK_STREAM;
        break;
    case SOCK_DGRAM:
        type = SOCK_DGRAM;
        break;
    case SOCK_RAW:
        type = SOCK_RAW;
        break;
    default:
        type = SOCK_STREAM;
        break;
    }
    if ((hp = gethostbyname(hostname)) == NULL) { /* do we know the
        host's */
        perror("gethostbyname");
    }
    bzero((char *) &sa, sizeof(sa));
    bcopy(hp->h_addr, (char *) &sin.sin_addr, hp->h_length); /*
        set address */
    sin.sin_family = AF_INET;
    sin.sin_port = htons((u_short) port);
    bzero((char *) &sa, sizeof(sa));
    sa.sin_family = AF_INET;
    sa.sin_addr.s_addr = htonl(INADDR_ANY);
    sa.sin_port = htons(0);
}

SOCKET::~SOCKET()
{
    close(sockfd);
    close(conct);
}

/*
 * server calls socket
 * parameter: none
 */
int SOCKET::sev_callsock()
{
    if ((sockfd = socket(AF_INET, type, 0)) < 0) /* create socket */
    {
        perror("SOCKET::sev_callsock(): can't open socket");
        return(-1);
    }
if (bind(sockfd, (struct sockaddr *) &:sa, sizeof(sa)) < 0) {
  herror("SOCKET::sev_callsock(): can't bind local address");
  close(sockfd);
  return(-1); /* bind address to
socket */
}
if(type == SOCK_STREAM) {
  listen(sockfd, 5);
}
return (sockfd);

int SOCKET::sev_accept()
{
  int i;
  i = sizeof(sin);
  if((conct = accept(sockfd, (struct sockaddr *)&sin, &i)) < 0)
  {
    herror("SOCKET::sev_accept(): can't accept connection");
    return (0);
  }
  return (conct);
}

/* cln_callsock: client calls socket
 * parameter: none
 */
int SOCKET::cln_callsock()
{
  int tries=4;
  if ((sockfd = socket(AF_INET, type, 0)) < 0) /* get socket */
  {
    herror("SOCKET::cln_callsock0: can't open socket");
    return(-1);
  }
  if(type == SOCK_STREAM) { /* using TCP protocol */
    while(1) { // let's try 5 times to connect server
      if (connect(sockfd, (struct sockaddr *) &sin, sizeof(sin))
        == -1) /* connect */
        {
        if(--tries <=0 ) {
          char host[32];
          gethostname(host, 32);
          cerr<<"CLIENT: can't connect to server , my
name is "<<host << endl;
          close(sockfd);
          return(0);
          }
          sleep(1);
          } else break;
    }
else { /* using UDP protocol */
    if (bind(sockfd, (struct sockaddr *) &sa, sizeof(sa)) < 0)
        herror("SOCKET::cln_callsock(): can't bind local
    address");
    close(sockfd);
    return(-1);
}

return(sockfd);

int SOCKET:recvint(int fd, int *in)
{
    int count;
    if((count = read(fd, in, sizeof(int))) < 0)
        return (-1);
    *in = ntohl(*in);
    return count;
}

int SOCKET:rsendint(int fd, int *w)
{
    int count;
    *w = htonl(*w);
    if((count = write(fd, w, sizeof(int))) < 0)
        return (-1);
    return count;
}

int SOCKET:recvdata(int sk, void *msg, int len)
{
    int nread, left, start=0;
    int lln = sizeof(sin);
    left = len;
    switch(type){
        case SOCK_STREAM: {
            while(left > 0) {
                if(left > MAXLENGTH){
                    nread = read(sk, msg, MAXLENGTH);
                }
                else nread = read(sk, msg, left);
                if(nread < 0)
                    return (nread);
                else if(nread == 0)
                    break;
            } /* while */
            break;
        }
    } /* switch */
}
left -= nread;
  msg += nread;
}
return (len - left);
break;
}
case SOCK_DGRAM: {
  nread = recvfrom(sk, msg, MAXLENGTH, 0, (sockaddr *)
  &sin, &lln);
  if(nread < 0) {
    perror("recvfrom error!");
    return (-1);
  }
  break;
}
default: break;
}
return (nread);

int SOCKET::senddata(int sk, void *msg, int len)
{
  int nwritten, left, start=0;
  int lln = sizeof(sin);
  left = len;
  switch(type){
    case SOCK_STREAM: {
      while(left >= 0) {
        if(left > MAXLENGTH) {
          nwritten = write(sk, msg, MAXLENGTH);
        }
        else nwritten = write(sk, msg, left);
        if(nwritten <= 0) return (nwritten);
        left -= nwritten;
        msg += nwritten;
      }
      return (len - left);
      break;
    }
    case SOCK_DGRAM: { 
      if((nwritten = sendto(sk, msg, len, 0, (sockaddr *)
      &sin, lln)) <0) 
      { 
        char host[32];
        gethostname(host, 32);
        cerr<<"Sendto error in " << host << endl;
        return (-1);
      }
      break;
    }
    default: break;
  }
  return (nwritten);
#ifndef /* SOCKET_C */

#include <stdio.h>
#include <iostream.h>
#include <string.h>
#include <stdlib.h>

void itoa(int, char *);

void itoa(int n, char *a)
{
    int b=1, x=1, c=0, i=0;
    while ((n/x)>=10){
        b++;
        x=x*10;
    }
    while (i<b) {
        c = n/x;
        n = n%x;
        x= x/10;
        a[i] = c+48;
        i++;
    }
}

/****************************
*** Utility Functions for SPIDER II
****************************/

#include <stdio.h>
#include <stdlib.h>
#include "profile.h"

void sendflag(int flag)
{
    int i;
    ofstream outfile(Task_Flag, ios::out);
    if(!outfile) {
        cerr << "No flag to update !!! \n";
        exit(-1);
    }
    outfile << flag << endl;
    outfile.close();

    116
int getflag()
{
    int flag;
    fstream file_fd;

    file_fd.open(Task_Flag, ios::in);
    if (!file_fd) {
        fprintf(stderr, "Can't fine the flag");
        return 0;
    }

    file_fd >> flag;

    return flag;
}

void itoa(int n, char *a)
{
    int b=1, x=1, c=0, i=0;

    while ((n/x)>=10){
        b++;
        x=x*10;
    }
    while (i<b) {
        c = n/x;
        n = n%x;
        x= x/10;
        a[i] = c+48;
        i++;
    }
}

void getname(char *a)
{
    char temp[30], i=0;

    gethostname(temp, 30);

    while (temp[i]!='.') {
        a[i]=temp[i];
        i++;
    }
}
// ********** qsort.C **********
// Quick Sort for Spider II
// client program
// **********……………………

#include <iostream.h>
#include <stdio.h>
#include "timer.c"
#include "OSB.C"

class Qsort
{
  public:
    Qsort();
    void dqsort(int size, int need);
  private:
    void call_OSB();
    void printing();
    void sort(int *a, int *temp, int left, int right);
    void merge (int *a, int *temp, int LeftPos, int RightPos, int RightEnd);
    void quick_sort (int first, int last);
  
  int num_need, s, num;
  int array[MAX];
  int a2[MAX];
};

Qsort::Qsort()
{
}

void Qsort::dqsort(int size, int need)
{
  s = size;
  num_need = need;
  call_OSB();
  cout << "final sort" << endl;
  // sort(&array[0],a2, 0,num-1);
  quick_sort(0, num-1);
  printing();
}

void Qsort::sort(int *a,int *temp, int left, int right)
{
  int Center;

  if (left < right)
  {
    Center = (left + right)/2;
    sort(a, temp, left, Center);
    sort(a, temp, Center+1, right);
  }
void Qsort::merge (int *a, int *temp, int LeftPos, int RightPos, int RightEnd)
{
    int LeftEnd = RightPos - 1;
    int TempPos = LeftPos;
    int NumElements = RightEnd - LeftPos + 1;

    while( LeftPos <= LeftEnd && RightPos <= RightEnd )
    {
        if( a[LeftPos] <= a[RightPos])
            temp[TempPos++] = a[LeftPos++];
        else
            temp[TempPos++] = a[RightPos++];
    }

    while(LeftPos <= LeftEnd)
    temp[TempPos++] = a[LeftPos++];

    while(RightPos <= RightEnd)
    temp[TempPos++] = a[RightPos++];

    for(int i = 0; i < NumElements; i++, RightEnd--)
        a[RightEnd] = temp[RightEnd];
}

void Qsort::quick_sort(int first, int last)
{
    int temp;
    if (first < last)
    {
        int pivot = array[first];
        int i = first;
        int j = last;
        while (i < j)
        {
            while (array[i] <= pivot && i < last)
                i += 1;
            while (array[j] >= pivot && j > first)
                j -= 1;
            if (i < j) //swap(array[i],array[j])
            {
                temp = array[i];
                array[i] = array[j];
                array[j] = temp;
            }
            //swap(array[j],array[first])
        }
    }
temp = array[first];
array[first] = array[j];
array[j] = temp;

quick_sort(first, j-1);
quick_sort(j+1, last);
}
}

void Qsort::call_OSB()
{
char a;
OSB *osb;
char *service;

for (int i=0; i<s; i++)
array[i] = rand()%10000;

service= "qs_slv";
osb = new OSB;

osb->spider_spawn(service, num_need);
osb->send_data(&array[0], s);
num = osb->get_return(&array[0]);
}

void Qsort::printing()
{
    // cout << "got after sort :
    // for(int i=0; i<num; i++) {
    //    cout << array[i] <<" 
    // }
    // cout << endl;
    cout << num << "Number Sorted" << endl;
}

void main(int argc, char *argv[])
{
    int size, n;
    if (argc <= 2) {
        cout << "please input sorting size and " << endl;
        cout << "how many workstations you like to use !!" << endl;
        exit(0);
    } else {
        if ((size = atoi(argv[1])) > MAX) {
            cout << "Sorting size can not larger the 1000000" << endl;
            exit(0);
        }
n = atoi(argv[2]);
start_time();
Qsort *qsort;
qsort = new Qsort;
qsort -> dqsort(size, n);
prn_time();
exit(0);

II -k-k-k-k-k-k-k-k^-k gg gIv" C
//Quick Sort for Spider II
// Slave program
// *******************

#include <iostream.h>
#include <stdio.h>
#include "osb_slv.C"

class Qs_slv
{
public:
   Qs_slv();
   void start_osb(int myport);
   void quick_sort(int first, int last);
   void printing();

private:
   int port;
   osb_slv *osb;
   int arr[MAX];
   int arr_size;
};
Qs_slv::Qs_slv()
{
}
void Qs_slv::start_osb(int myport)
{
   port = myport;
   osb = new osb_slv(port);
```cpp
arr_size = osb->get_data(&arr[0]);
quick_sort(0, arr_size-1);
// printing();
osb->return_data(arr, arr_size);
}

void Qs_slv::quick_sort(int first, int last)
{
  int temp;
  if (first < last)
  {
    int pivot = arr[first];
    int i = first;
    int j = last;
    while (i < j)
    {
      while (arr[i] <= pivot && i < last)
        i += 1;
      while (arr[j] >= pivot && j > first)
        j -= 1;
      if (i < j) //swap(arr[i],arr[j])
        {
          temp = arr[i];
          arr[i] = arr[j];
          arr[j] = temp;
        }
    }
    //swap(arr[j],arr[first])
    temp = arr[first];
    arr[first] = arr[j];
    arr[j] = temp;
    quick_sort(first, j-1);
    quick_sort(j+1, last);
  }
}

void Qs_slv::printing()
{
  cout << "array after sort : ";
  for (int i=0; i < arr_size; i++) {
    cout << arr[i] << " ";
  }
  cout << endl;
}

main( int argc, char *argv[])
{
  int port;
```
Qs_slv *slv;

if (argc <=1) {
    fprintf(stderr, " Port not found ");
    exit(0);
}

else {
    port = atoi(argv[1]);
    slv = new Qs_slv();
    slv->start_osb(port);
}

// **********prime.C ************
// Prime Search for Spider II
// *****************************

#include <iostream.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "timer.c"
#include "OSB.C"

class Prime
{
    public:
        Prime();
        ~Prime();
        void psearch(int size, int numofslv);

    private:
        int arr[MAX], arrsize, num;
        char *service;
        void print();
        void call_OSB();
};

Prime::Prime()
{
    arrsize = 0;
    service = "ps_slv";
}

Prime::~Prime()
void Prime::rpsearch(int size, int numofslv)
{
    int i, j, b, n;
    num = numofslv;
    b = num*2;
    n = size/(num*2);
    j = 0;
    for (i = 0; i < num; i++) {
        arr[j++] = i*n + 1;
        arr[j++] = (i+1)*n;
        arr[j++] = (b-i-1)*n + 1;
        arr[j++] = (b-i)*n;
    }
    arrsize = j;
    cout << "arrsize" << arrsize << endl;
    cout << "arr = ";
    for (i = 0; i < arrsize; i++)
        cout << arr[i] << " , ";
    cout << endl;
    call_OSB();
}

void Prime::call_OSB()
{
    OSB *osb;
    osb = new OSB;
    osb->spider_spawn(service, num);
    osb->send_data(&arr[0], arrsize);
    num = osb->get_return(&arr[0]);
    cout << "num = " << num << endl;
    print();
}

void Prime::print()
{
    int i;
    cout << "arr = " << endl;
    cout << "prime number = " << num << endl;
}

void main(int argc, char *argv[])
{
    int size, numofproc;
    double t;

    if (argc <= 2) {
        cout << "please input prime search size and " << endl;
        cout << "how many workstations you like to use !!" << endl;
        exit(0);
    }
else {
    size = atoi(argv[1]);
    numofproc = atoi(argv[2]);
    start_time();
    Prime *prim;
    prim = new Prime;
    prim -> psearch(size, numofproc);
    prim_time();
    t = get_time();
    cout << "time = " << t << endl;
}

// ********** qs_slv.C ********
// Quick Sort for Spider II
// Slave program
// ***********************

#include <iostream.h>
#include <stdio.h>
#include "osb_slv.C"

class Qs_slv
{
    public:
        Qs_slv();
        void start_osb(int myport);
        void quick_sort(int first, int last);
        void printing();

    private:
        int port;
        osb_slv *osb;
        int arr[MAX];
        int arr_size;

};

Qs_slv::Qs_slv()
{
}

void Qs_slv::start_osb(int myport)
{
    port = myport;

    osb = new osb_slv(port);
arr_size = osb->get_data(&arr[0]);
quick_sort(0, arr_size-1);
// printing();
osb->return_data(arr, arr_size);

}

void Qs_slv::quick_sort(int first, int last)
{
  int temp;
  if (first < last)
  {
    int pivot = arr[first];
    int i = first;
    int j = last;
    while (i < j)
    {
      while (arr[i] <= pivot && i < last)
        i += 1;
      while (arr[j] >= pivot && j > first)
        j -= 1;
      if (i < j) //swap(arr[i],arr[j])
      {
        temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
      }
    } //swap(arr[j],arr[first])
    temp = arr[first];
    arr[first] = arr[j];
    arr[j] = temp;
    quick_sort(first, j-1);
    quick_sort(j+1, last);
  }
}

void Qs_slv::printing()
{
  cout << "array after sort : ";
  for (int i=0; i < arr_size ; i++) {
    cout << arr[i] << " ";
  }
  cout << endl;
}

main( int argc, char *argv[])
{
  int port;

}
Qs_slv *slv;

if (argc <= 1) {
    fprintf(stderr," Port not found ");
    exit(0);
}

else {
    port = atoi(argv[1]);
    slv = new Qs_slv();
    slv->start_osb(port);
}

// *********** Matrix_p.C ***********
// Matrix Multiplication for Spider II
// Client side program
// ****************************************

#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include "timer.c"
#include "OSB.C"

class Matrix_p
{
    public:
        Matrix_p(int size);
        void dmatrix();
        void print();
        void call_OSB();
        void after_distribution();

    private:
        int matrix1[900][900], matrix2[900][900];
        int arr1[MAX], arr2[MAX], arr3[MAX], data[9][MAX];
        int size, count1, count2, newsize;
        char *service;
    }

Matrix_p::Matrix_p(int s)
{
    int i, j, c=0, d=0;
    size = s;
    count1 =0;
    count2 =0;
    service = "matrix_srv";
for (i=0; i<9; i++)
    bzero(data[i], sizeof(data[i]));

for (i=0; i<size; i++) {
    for (j=0; j<size; j++) {
        matrix1[i][j] = rand()%100;
        matrix2[i][j] = rand()%100;
        if (matrix1[i][j] > 80)
            matrix1[i][j] = -(matrix1[i][j]);
        if (matrix2[i][j] < 20)
            matrix2[i][j] = -(matrix2[i][j]);
    }
}

void Matrix_p::dmatrix()
{
    int dsize, send_array_size, total_dsize;
    int i, j, X, Y;

    dsize=size/3;
    total_dsize = dsize*dsize;
    send_array_size = size*size;

    for (i=0; i<3; i++)
        for (j=0; j<3; j++)
            for (X=(0+dsize*i); X<(dsize+dsize*i); X++)
                for (Y=(0+dsize*j); Y<(dsize+dsize*j); Y++){
                    arrl[count1++]=matrixl[X][Y];
                    arr2[count2++]=matrix2[X][Y];
                }

call_OSB();
}

void Matrix_p::call_OSB()
{
    OSB *osb;
    int num=9;

    osb = new OSB;
    osb->spider_spawn(service, num);
    osb->send_data_matrixp(&arrl[0], &arr2[0], count1);
    newsize = osb->get_return_matrixp(data);

    cout << "newsize = " << newsize << endl;

    after_distribution();
}

void Matrix_p::after_distribution()
{
    int i, j, X, Y, counter, block_size;

    for (i=0; i<9; i++)
        bzero(data[i], sizeof(data[i]));

    for (i=0; i<size; i++) {
        for (j=0; j<size; j++) {
            matrix1[i][j] = rand()%100;
            matrix2[i][j] = rand()%100;
            if (matrix1[i][j] > 80)
                matrix1[i][j] = -(matrix1[i][j]);
            if (matrix2[i][j] < 20)
                matrix2[i][j] = -(matrix2[i][j]);
        }
    }

    void Matrix_p::dmatrix()
    {
        int dsize, send_array_size, total_dsize;
        int i, j, X, Y;

        dsize=size/3;
        total_dsize = dsize*dsize;
        send_array_size = size*size;

        for (i=0; i<3; i++)
            for (j=0; j<3; j++)
                for (X=(0+dsize*i); X<(dsize+dsize*i); X++)
                    for (Y=(0+dsize*j); Y<(dsize+dsize*j); Y++){
                        arrl[count1++]=matrixl[X][Y];
                        arr2[count2++]=matrix2[X][Y];
                    }

        call_OSB();
    }

    void Matrix_p::call_OSB()
    {
        OSB *osb;
        int num=9;

        osb = new OSB;
        osb->spider_spawn(service, num);
        osb->send_data_matrixp(&arrl[0], &arr2[0], count1);
        newsize = osb->get_return_matrixp(data);

        cout << "newsize = " << newsize << endl;

        after_distribution();
    }

    void Matrix_p::after_distribution()
    {
        int i, j, X, Y, counter, block_size;

        for (i=0; i<9; i++)
            bzero(data[i], sizeof(data[i]));

        for (i=0; i<size; i++) {
            for (j=0; j<size; j++) {
                matrix1[i][j] = rand()%100;
                matrix2[i][j] = rand()%100;
                if (matrix1[i][j] > 80)
                    matrix1[i][j] = -(matrix1[i][j]);
                if (matrix2[i][j] < 20)
                    matrix2[i][j] = -(matrix2[i][j]);
            }
        }

        Solution
dsize=9;
        for (i=0; i<9; i++)
            for (j=0; j<9; j++)
                matrix1[i][j] = rand()%100;

    Solution
    
    Solution
bzero(arr1, sizeof(arr1));
bzero(arr2, sizeof(arr2));
bzero(arr3, sizeof(arr3));

newsize=newsize-1;
// cout << "newsize = " << newsize << endl;

for (i=0; i< 9;i++) {
    if((data[i][newsize]==0) || (data[i][newsize]==1) ||
       (data[i][newsize]==2)) {
        for (j=0; j< newsize; j++)
            arr1[j]=arr1[j]+data[i][j];
    }
    if((data[i][newsize]==3) || (data[i][newsize]==4) ||
       (data[i][newsize]==5)) {
        for (j=0; j< newsize; j++)
            arr2[j]=arr2[j]+data[i][j];
    }
    if((data[i][newsize]==6) || (data[i][newsize]==7) ||
       (data[i][newsize]==8)) {
        for (j=0; j< newsize; j++)
            arr3[j]=arr3[j]+data[i][j];
    }
}

 block_size = size/3;

counter = 0;
for (i=0; i<3; i++){
    for (X=0; X < block_size; X++){
        for (Y=0+block_size*i; Y < block_size+block_size*i; Y++){
            matrix1[X][Y] = arr1[counter];
            matrix1[X+block_size][Y] = arr2[counter];
            matrix1[X+block_size*2][Y] = arr3[counter++];
        }
    }
}

void Matrix_p::print()
{
    int i, j, c=0, d=0;
}

void main(int argc, char *argv[])
{
    int size, n, m;
    if (argc <= 1) {
        cout << "please input Matrix size " << endl;
        exit(0);
    }
else {
    if ((size = atoi(argv[l])) > 900) {
        cout << " size can not larger the 900" << endl;
        exit(0);
    }
    while ((m = size%3) != 0)
        size = size--;
    cout << "size = " << size << endl;
    Matrix_p *matrixp;
    matrixp = new Matrix_p(size);
    start_time();
    matrixp -> dmatrix();
    prn_time();
    exit(0);
}

II ****************** matrix_srv.C **************
// Distributed Matrix Multiplication for Spider II
// server side program
II **********************************************

#include <iostream.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "osb_slv.C"

class Matrix_srv
{
    public:
        Matrix_srv();
        void start_osb(int port);

    private:
        int port;
        osb_slv *osb;
        int arr1[MAX], arr2[MAX];
        int A[300][300], C1[300][300], C2[300][300], C3[300][300];
        int B[300][300], B2[300][300], B3[300][300];
        int outcome[MAX];
        int arr_size1, arr_size2, index, matrix_size;
        void print();
        void matrix_multi();
        void return_to_osb();
};

Matrix_srv::Matrix_srv()
{ index = 0;
void Matrix_srv::start_osb(int myport)
{
    int i, b, x, y;
    int arr1[MAX], arr2[MAX];
    double tmp;
    port = myport;

    osb = new osb_slv(port);
    arr_size1 = osb->get_data_matrixp(&arr1[0], &arr2[0]);
    arr_size2 = 3*arr_size1;

    tmp = sqrt(arr_size1);
    matrix_size = (int) tmp;

    i=0;
    for (x=0; x< matrix_size; x++)
        for (y=0; y< matrix_size; y++)
            A[x][y]=arr1[i++];

    i=0;
    for (x=0; x< matrix_size; x++)
        for (y=0; y< matrix_size; y++)
            B1[x][y]=arr2[i++];

    for (x=0; x< matrix_size; x++)
        for (y=0; y< matrix_size; y++)
            B2[x][y]=arr2[i++];

    for (x=0; x< matrix_size; x++)
        for (y=0; y< matrix_size; y++)
            B3[x][y]=arr2[i++];

    matrix_multi();
    return_to_osb();
}

void Matrix_srv::matrix_multi()
{
    int x, y, i;

    for (x=0; x < matrix_size; x++)
    for (y=0; y < matrix_size; y++)
        for (i=0; i< matrix_size; i++) {
            C1[x][y]=C1[x][y]+A[x][i]*B1[i][y];
            C2[x][y]=C2[x][y]+A[x][i]*B2[i][y];
            C3[x][y]=C3[x][y]+A[x][i]*B3[i][y];
        }
}

void Matrix_srv::return_to_osb()
int x, y, count = 0;
bzero(arr1, sizeof(arr1));

for (x=0; x< matrix_size; x++) {
    for (y=0; y< matrix_size; y++)
        arr1[count++]=C1[x][y];
}
for (x=0; x< matrix_size; x++) {
    for (y=0; y< matrix_size; y++)
        arr1[count++]=C2[x][y];
}
for (x=0; x< matrix_size; x++) {
    for (y=0; y< matrix_size; y++)
        arr1[count++]=C3[x][y];
}

osb->return_data_matrixp(arr1, count);

}

void Matrix_srv::print()
{
    int i;
    for (i=0;i<index;i++)
        cout << arr1[i] << " ", ";
    cout << endl;
}

void main (int argc, char *argv[])
{
    int port;

    Matrix_srv *srv;

    if (argc <=1) {
        fprintf(stderr," Port not found ");
        exit(0);
    }
    else {
        port = atoi(argv[1]);
        srv = new Matrix_srv();
        srv->start_osb(port);
    }
}
REFERENCES

http://www.beowulf.org/intro.html


http://www-unix.mcs.anl.gov/mpi/

http://www.cs.huji.ac.il/mosix/

http://www.beowulf-underground.org/doc_project/HOWTO/english/Beowulf-HOWTO.html
