1997

Spider: An overview of an object-oriented distributed computing system

Han-Sheng Yuh

SPIDER: AN OVERVIEW OF AN OBJECT-ORIENTED DISTRIBUTED COMPUTING SYSTEM

A Thesis

Presented to the

Faculty of

California State University,

San Bernardino

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

in

Computer Science

---

by

Han-Sheng Yuh

June 1997

SPIDER: AN OVERVIEW OF AN OBJECT-ORIENTED DISTRIBUTED COMPUTING SYSTEM

A Thesis

Presented to the

Faculty of

California State University,

San Bernardino

---

by

Han-Sheng Yuh

June 1997

Approved by:

[redacted]

Dr. Arturo I. Concepcion, Chair, Computer Science — 12 Jun 97 Date

[redacted]

Dr. Tong L. Yu

[redacted]

Dr. Kay Zemoudeh

ABSTRACT

Parallel and distributed computing on networks of workstations has been gaining more attention in recent years. Clusters of workstations connected with high-speed networks can have computational speed approaching that of supercomputers. The Spider Project is an object-oriented distributed system which provides a testbed for researchers in the Department of Computer Science, CSUSB, to conduct research on distributed systems. The object-oriented approach was used because of easy maintenance, modification, and simplicity in adding more features and functionalities to Spider in the future.

In this thesis we have derived the specification and design of the Spider distributed system by studying well known distributed systems: Sun's Spring Project, OSF's DCE, Oak Ridge National Laboratory's PVM, and University of Wisconsin-Madison's Condor Project. We identified the functionalities of the Spider system which are: distributed file system, security, clock synchronization, scheduling management and distributed computation. To illustrate the validity of the specification and design of Spider, the functionality of distributed computation was implemented and the performance of this implementation was analyzed and compared with Parallel Virtual Machine (PVM) and SGI

Challenge supercomputer. A graphics user interface was also implemented using Java applets, so that Spider can be accessed on the Internet.

ACKNOWLEDGEMENTS

I would like to thank Dr. A. I. Concepcion, Dr. Tong L. Yu, and Dr. Kay Zemoudeh for the very helpful knowledge of the distributed and parallel systems. Al Geist, the manager of the PVM team, for his help in answering all questions about PVM. Associated Students Incorporated at CSUSB provides an ASI research fund for my thesis research.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1 Introduction

1.1 Introduction

Parallel and distributed computation on networks of workstations has been gaining more attention in recent years. Most current commercial workstations offer better price and performance, and high-speed switch-based networks have higher bandwidths than before and have significant improvements in reliability. Such advantages provide the necessary environment for developing distributed systems. A distributed system is a system that is a collection of computers which run their own operating systems or distributed operating system without having a global memory or a single clock, and computers communicate with each other by exchanging messages over a network. A distributed system not only can provide sharing of expensive resources, such as laser printers and disk drives, but also offer users powerful computation capability.

A distributed system can include distributed operating system, distributed file system, distributed scheduling, distributed shared memory, etc. These functionalities support transparency where users do not need to worry about the location of resources. To achieve distributed computing on a distributed system, users do not need to know how the

program will be executed and where it will take place. The system will take responsibility to handle jobs distribution and migration, if necessary. Nowadays, most organizations have high-speed local area networks (LAN) interconnecting many general-purpose workstations, the combined computational resources may exceed the power of a single high-performance computer or supercomputer. Thus, a distributed system is able to combine the computational power of all workstations into one huge virtual computer (see Figure 1.1.) A user can make use of all resources in the whole distributed system.

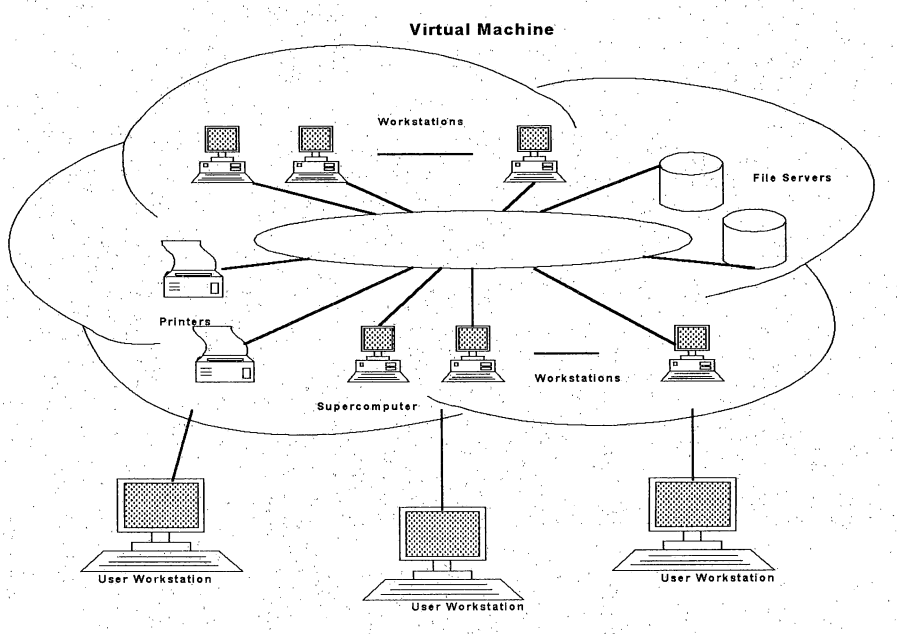

Figure 1.1 A distributed system as a virtual machine.

The following are issues which are important in the design of a distributed system:

- *Transparency* is the major advantage of a distributed system. Users should not be aware of the location of service/data because it is invisible (transparent) to them [23]. The user will receive the result displayed on his/her workstation where the program is launched. It is also possible that the program is executed in the local machine but this is hidden from the user. A distributed system must provide an efficient way to deliver the services to users.
- *Coherence.* Since there is no global clock and global memory in a distributed system, a process can obtain a coherent but partial view of the system or a complete but incoherent view of the system [21]. A distributed system must provide a consistent global state for the system.
- *Fault Tolerance.* A distributed system must provide a way to recover whole or partial process if one processor is failed during program execution, the user does not need to know this situation happened.
- *Concurrency Control* is another main problem dealing with database and file systems. A distributed system must provide a protocol (e.g., single-writer/multiple-

readers) for users accessing consistent data, whether timestamp or lock algorithms [21, 23] may be applied.

- *Heterogeneity.* There are a variety of hardware platforms in the market. A distributed system must provide the same service to every kind of platform.

The above issues are necessary to be considered and discussed before developing a distributed system, and also they are the goals for a distributed system.

1.2 Motivations

Powerful workstations and higher network bandwidths have provided a crucial environment for researching and developing distributed and parallel systems. This new development gave us the motivations for undertaking this study.

The motivations for the Spider Project are:

a) To conduct research in distributed and parallel systems.

b) To provide a testbed for students and faculty to do related research on distributed systems.

c) To build a distributed system, which will be able to make full use of the workstations in the Department of computer Science, at California State University at San Bernardino (CSUSB).

The overview of the Spider Project will be explained in this thesis. Why do we call this project *Spider*? We imagine the distributed operating system as a spider and the whole networked distributed system as a web. A spider on the web can move to any place quickly (transparency), and it won't affect the whole web if a part of web is broken (fault tolerance).

1.3 Thesis Goals

The Spider Project is the first distributed system study in the Department of Computer Science at CSUSB. The Spider Project will be an object-oriented distributed system which will provide a testbed for users to conduct research on distributed and parallel systems. Currently, there are 24 Data General Aviion workstations, 40 Sun SparcClassic workstations, 30 Silicon Graphics Iris Indigo advanced graphics workstations, 24 X-terminal workstations, 30 IBM compatible PCs in the Department of Computer Science at CSUSB. Furthermore, there are two supercomputers, a Silicon Graphics Power Challenge XL with 1.2 GFLOP peak performance and an Intel iPSC 860 16-node hypercube. In order to support real graphics, the Challenge XL is connected to 10 Indigo workstations using a Fore ATM ASX-200 switch. Thus, the computing power of the Department of Computer Science at

CSUSB will be greatly enhanced if there is a distributed system.

Why should the Spider Project be built and design in an object-oriented approach? Object orientation technology has emerged in the last few years and became a good methodology for software and systems from design to implementation to maintenance. The object-oriented approach (OOA) has the ability to provide modularity, abstraction, and information hiding. The idea of abstraction in OOA is to distill the essence of a problem to understand it better. Furthermore, future work will be made easy to modify or to add functionalities to this system. Therefore, the Spider Project will be designed in an object-oriented approach. The C++, a popular object-oriented language, will be the primary programming language for all implementations.

One of the main benefits of object-oriented system development is maintainability. System developers can easily modify, upgrade, and expand services of the system by changing the characteristics of an object(s), or replacing a new object without affecting other components (objects). All external objects will communicate with objects. Internal data and structures can be refined without impacting other parts of the system. Since each object has a set of methods, objects can be reused to enhance developer productivity. The

Spider Project will adopt the object-oriented approach methodology to develop an object-oriented distributed system.

The goals of this thesis for the Spider Project are the following:

a) Study Spring [14], DCE [17], PVM [7], and Condor [13, 24]. The first two are distributed systems, the third one is a distributed computing system, and the last one is a scheduling system. They will be examined and studied in order to provide the background for specifying and designing the Spider System.

b) Specify and design the structure and functionalities of the Spider System. The functionalities must render service to a user in a transparent and distributed manner. The functionalities will be identified in this thesis.

c) Implement one of the identified functionalities. In order to show that the specification and design are valid, one of the identified functionalities will be implemented.

d) The implementation of the selected functionality in (c) must be highly maintainable in order to extend or add more services into this functionality.

As mentioned earlier, there are many powerful workstations in the Department of Computer Science at CSUSB, but all of them are traditional time-sharing systems. The Spider Project will harness all of these powerful resources into a single virtual machine. Each physical machine will be fully utilized under the Spider object-oriented distributed system. With the Spider, every user can access all the resources in the system locally or remotely.

1.4 Organization of Thesis

This thesis is organized into five chapters. In Chapter 1, the introduction and motivation of the Spider Project are mentioned. In Chapter 2, the Sun's Spring, OSF's DCE, Oak Ridge National Laboratory's PVM, and Condor System will be discussed and explained in a survey for specifying and designing the Spider distributed system. In Chapter 3, the functionalities of the Spider System will be specified and designed in detail. In Chapter 4, the implementation of one of the functionalities, which is Distributed Computation Service, for Spider will be discussed. The last chapter will discuss future works and directions.

CHAPTER 2 Related Works

To define the structure and functionality for the Spider system, we need to survey existing systems and learn from their strengths and weaknesses. The Sun Microsystems' Spring distributed operating system [14] and the Open Software Foundation's Distributed Computing Environment (OSF's DCE) [17] are two distributed systems designed in object-oriented approach. They provided good references for the design of the Spider system. The Parallel Virtual Machine system (PVM)[7] is a software system that provides a virtual machine to a single user by collecting a set of heterogeneous UNIX computers and the user's programs can be executed in a distributed and/or parallel manner. The Condor scheduling system [13, 24] is designed for a workstation environment so that users can make full utilization of any available workstation for their processing needs. The Spider system will be based on concepts from these latter two examples in regards to distributed computation. The implementation of the distributed computation will relate closely to the design of PVM. The Condor system will be the background for the scheduling server of the Spider system.

This chapter will discuss major design structures of Spring, DCE, PVM, and Condor.

2.1 The Spring Distributed Operating System

The Spring Project [14] at Sun Microsystems Inc. is developing new technologies for constructing operating systems and for simplifying distributed programming. The Spring distributed operating system is constructed and applied to this project. The main design methodology of the Spring Project is in object-oriented approach because they want to build a highly modular, object-oriented operating system, which is focused around a uniform interface definition language (IDL) [16, 17]. They also want to innovate the current operating systems to be more open, extensible and flexible.

Sun decided that the Spring system should have a strong and explicit architecture: one that would pay attention to the interfaces between software components, which are really how a system's structure is expressed, but the interfaces do not provide much information on how they are implemented [14].

2.1.1 The Overview of the Spring System

The Spring system is a microkernel-based system. The microkernel consists of the nucleus and the virtual memory manager. The nucleus manages all inter-process communication and the virtual memory manager controls the memory

management hardware [14]. All other services are defined as objects in the user level and they are replaceable and substitutable. Figure 2.1 shows the major system components of the Spring system.

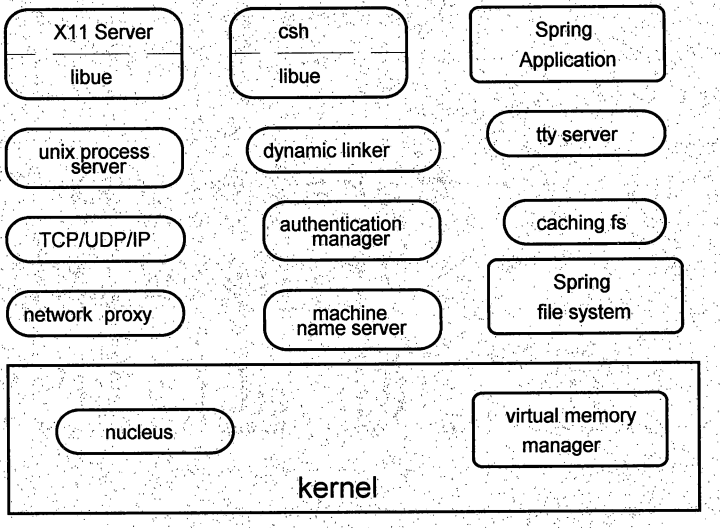

Figure 2.1 Major system components of a Spring node [14].

2.1.2 The Spring Nucleus

The microkernel of the Spring system is called nucleus. To reduce the performance loss by being split into different address spaces, the Spring nucleus is designed to provide a fast inter-process communication (IPC). It supports three basic abstractions: domains, threads, and doors [9].

2.1.2.1 Components of a Nucleus

a) Domains: Domains has the same form as processes in Unix and tasks in Mach [1]. They provide an address

space to execute and keep all resources information for each application, such as threads and doors, for applications.

b) Threads: Each domain can have multiple threads executed within it. All threads are accessible via cross-domain calls.

c) Doors: Doors can support calls between domains. It is similar to the ports in Mach [1]. A door is a particular entry point to a domain, represented by both a program counter and unique identifier assigned by the domain.

Each domain has a table of doors to which the domain has access. The user application uses door identifiers to reference doors. Door identifiers are mapped through the domain's door table into actual doors. In Spring, a door may be referenced by several different door identifiers in several different domains (see Figure 2.2).

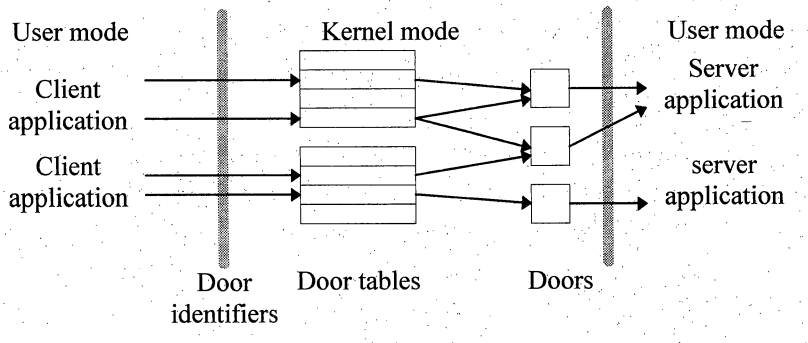

Figure 2.2 Doors and door tables [9].

2.1.2.2 Inter-Process Communication

To provide a fast-efficient IPC, Spring supports three different ways of door invocation and door returns [9].

a) Fast-path: When the door arguments are simple data values and size is less than 16 bytes, the fast-path calls and returns will be applied to achieve performance.

b) Vanilla-path: When the data is less than 5 Kbytes and some moderate number of doors are being passed as arguments or results, it will trap into the nucleus to copy data argument to the target domain.

c) Bulk-path: When the arguments or results contain large amount of data, the nucleus will use virtual memory manager re-mapping to transmit the data.

2.1.3 Subcontract

Subcontract is a flexible and extensible mechanism for plugging in different kinds of object runtime machinery which allows control over how object invocation is implemented, over how object references are transmitted and released, and similar object runtime operations [10].

It has become common to provide remote procedure calls (RPC) facilities that extend the semantics of local procedure calls to distributed systems. In an object-

oriented approach, RPC becomes the form of remote object invocation [14]. Due to the various RPC systems that each provides different application requirements, the Spring project uses *subcontracts*, which are replaceable modules, to give control of the basic mechanisms of object invocation and argument passing.

Subcontracts are separated from object implementations and object interfaces. It is easy for object implementers to either select and use an existing subcontract or to implement a new subcontract.

A Spring object is noticed by a client as consisting of three things [10]: 1) a method table, which contains an entry for each operation implied by the object's type definition; 2) a subcontract operations vector, which specifies the basic subcontract operations; 3) object's representation, which is some client-local private state.

Spring's components and objects are defined by strong interfaces that use the IDL [14]. By using IDL to define interfaces, developers won't be tied for any single programming language. An IDL interface can be compiled into three parts: client side stub, server side stub, and header file for the interface. Stubs generated from an IDL interface description can hide the object invocation from clients to deliver a transparent service.

The code of stubs can transform the method invocation into calls on either the object's regular method table or on its subcontract operation vector. Figure 2.3 shows the logical progression of a call to a server-based Spring object.

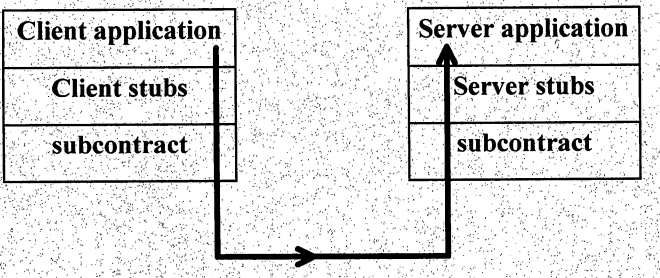

Figure 2.3 Invoking a method on a server-based object [10].

Different objects may need different subcontracts, but the basic principles of subcontracts are the same. The client-side subcontract has five basic operations: marshal, invoke, unmarshal, marshal_copy, and invoke_preamble. The server-side subcontract operations are creating a Spring object, processing incoming calls, and revoking an object [10].

The Spider system will use similar mechanisms for object invocation, which will be discussed in Chapter 3 and the implementation will be explained in Chapter 4.

2.1.4 File System

Spring file system [15] defines file objects, which inherit from the memory object and *io* interfaces, that are implemented by file servers. Thus, file objects can be memory mapped and accessed using read/write operations of the I/O interfaces. Spring file system uses the Spring security and naming architectures to provide access control and directory services.

The implementations of Spring file server consist of two systems: Spring Storage File System (SFS) and Caching File System (CFS). SFS is implemented using two layers, disk layer and coherency layer [15] (see Figure 2.4). The disk layer implements an on-disk UNIX compatible file system. The coherency layer is stacked on the disk layer and implements a per-block multiple-reader/single-writer coherency protocol. Also, the coherency layer keeps track of the state of each file object and of each cache object that holds the block at any point in time.

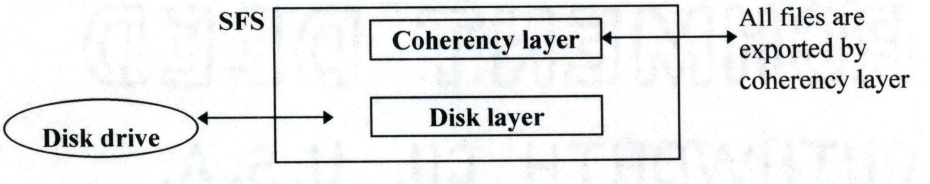

Figure 2.4 Spring SFS [15].

2.1.5 Security in Spring

To provide secure access to objects, Spring supports two basic mechanisms in security: Access Control Lists (ACL) and software capabilities.

An ACL defines which users of groups are allowed access to the particular objects. These ACL can be checked at runtime to determine whether a given client is really allowed to access a given object [14].

An object reference is created by the object's server, when the client proves that it is allowed to access that given object, that acts like a software capability. This object reference contains a nucleus door that points to a front object inside the object's server [14]. An object's server may create many different front objects, encapsulating different access rights, all pointing to the same underlying object. Thus, when the client assigns an object invocation on the object reference, the request can be securely transmitted to the front object. The front object checks the client's access right, if it is permissible then forwards the request into the server. Figure 2.5 shows the diagram of a client accessing a secure object.

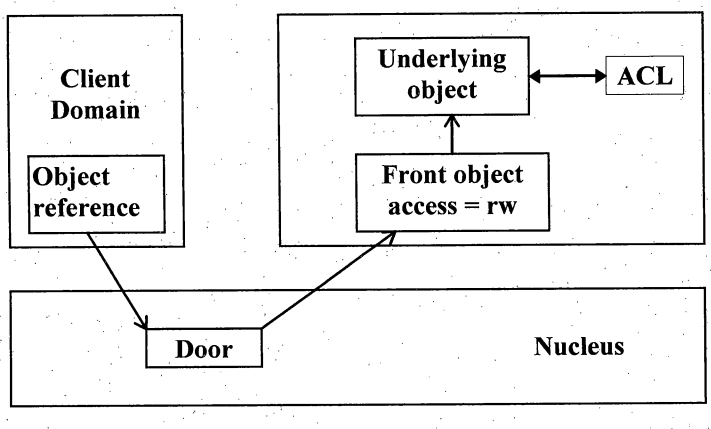

Figure 2.5 A client accessing a secure object [14].

When a client is given the object reference from the object's server, client can pass that reference to other clients. These other clients are able to have the same access rights as the original client and use the object reference freely.

2.1.6 Conclusions of the Spring Project

The Spring operating system is still an on-going project at Sun Microsystems Inc., but they have innovated the traditional operating systems to build one based on strong interfaces with openness and extensibility in an object-oriented approach. Sun hopes that the Spring system will replace Solaris in the future as a step toward a

distributed system environment. So far, the Spring operating system can only work on machines using Sun's architectures.

2.2 Distributed Computing Environment

The Open Software Foundation's (OSF) Distributed Computing Environment (DCE) is not like any other microkernel-based distributed systems, which are constructed from scratch. DCE is built on top of current operating systems. Since most environments (operating systems) today include technologies from a variety of vendors, DCE provides a common infrastructure for all kinds of systems [17].

Providing this common, multi-vendor infrastructure is the goal of the OSF's DCE. DCE provides the key services required for supporting distributed applications, including:

- support for remote procedure call (RPC) between clients and servers;
- a directory service to let clients find servers;
- security services to ensure access between clients and servers;
- a time service to synchronize the system clocks throughout the network;
- a threads service to provide multiple threads of execution capability; and

- distributed file services to provide access to files across a network.

2.2.1 The Overview of DCE

DCE supports the construction and integration of client/server applications in heterogeneous distributed environments. DCE has been designed to inter-work with existing standards in a number of areas. For example, a group of DCE machines can communicate with each other and with the outside world using either TCP/IP or the OSI protocols [21, 23]. User processes act as clients to access services provided by server processes, which can be local or remote. Figure 2.6 shows the various components of the DCE architecture.

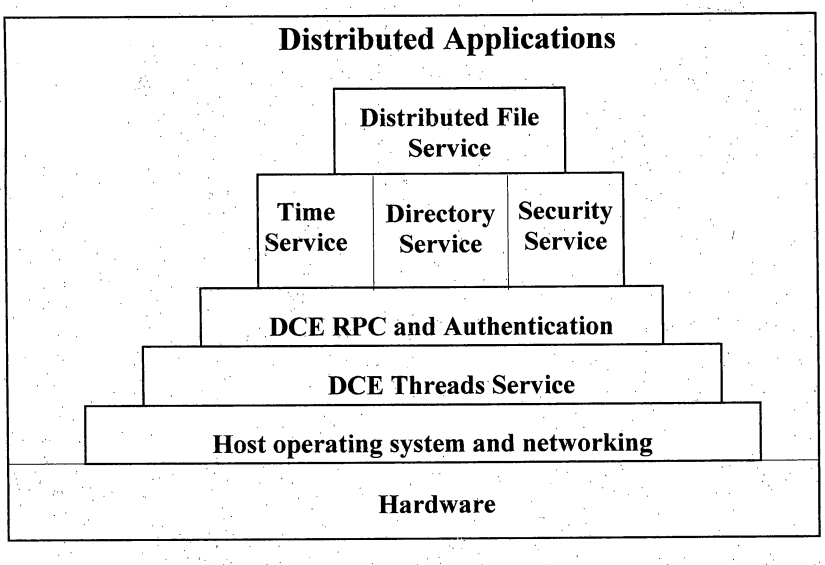

Figure 2.6 OSF DCE Architecture [23].

According to the four factors (purpose, security, overhead, and administration), users, machines, and other resources in a DCE system are grouped together to form cells [17]. All the DCE services are based upon these cells.

The following sections will discuss DCE's RPC, security service, and distributed file service.

2.2.2 Remote Procedure Call

DCE is based on the client/server model to provide a distributed computing environment. RPC dominates the communication in DCE. When requesting a service, Client makes an RPC to a remote server process. Before a client connects to a server's process via RPC, DCE handles all internal tasks, such as locating the server, binding to it and performing the call [23].

DCE RPC system hides all details of complex data transformations and communications from the user. The client only needs to make a local procedure call to receive a remote service. The intermediate procedures can be generated from an interface definition language (IDL) that is similar to the Spring's objects. A unique identifier is given when a IDL file is compiled. This identifier is for a client's process to locate a correct server, then the client is able

to receive correct information or services. Figure 2.7 shows the steps involved in binding client and server.

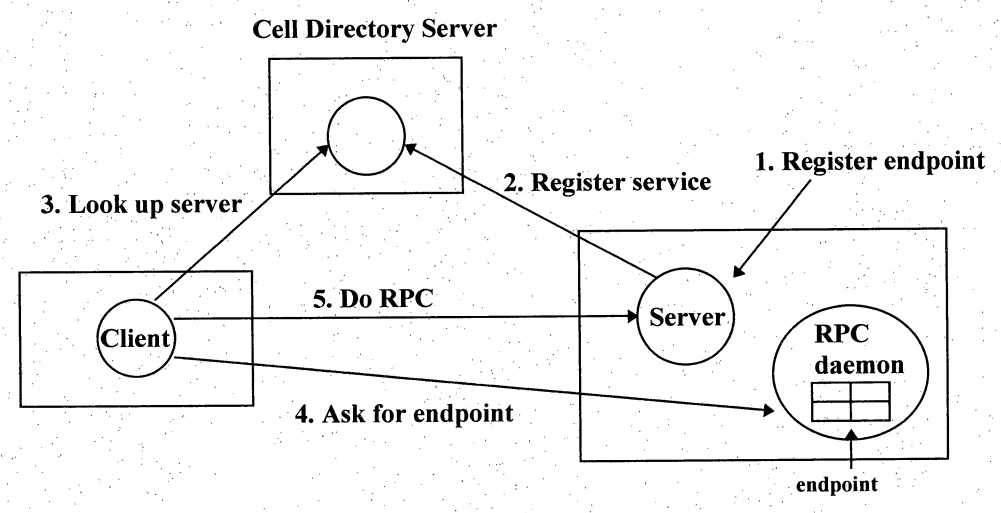

Figure 2.7 Client-to-server binding in DCE [23].

The endpoint, likes Spring's doors and Mach's ports, is a numerical address on the server's machine to which network connection can be attached and messages sent [23]. More discussions on RPC will be in Chapter 3.

2.2.3 Security Service

Security is always a major concern in a networked environment. In DCE, every user and process has its own *principal* when it needs to communicate securely [23]. DCE security can assign proper resources to each principal and

provide a secure cryptography for transmitting all information in an insecure network.

The major components of DCE security and their duties are the following [17, 23]:

a) Registry Server: the registry server manages the security data base, the registry, which contains the names of all principals, groups, and organizations.

b) Authentication Server: the authentication server verifies the claimed identity of the principal and grants a proper ticket that allows this principal to do other subsequent authentication without having to provide the password again.

c) Privilege Server: the privilege server issues PACs (Privilege Attribute Certificates) to authenticated user. The PAC is an encrypted information that has the principal's identity, group, and organization information, such that all servers can be instantly convinced without need for providing any additional information.

d) Login Facility: the login facility is a program using the authentication and privilege servers to provide all the necessary tickets and PACs when users are logging in.

Once a user is logged in a DCE system, the user can use a client program to access remote server process via authenticated RPC. In DCE, every resource has an ACL (Access Control List), which tells security who may access the resource. On the server side, when the application server receives the incoming request, the server verifies the requester's identity using the PAC and checks its ACL to see if the requester has the right to use the service or resource.

2.2.4 Distributed File System

The distributed file system (DFS) of DCE is to provide users and processes access to all files within a DCE system they are authorized to use. DFS has two main parts: local part and wide-area part. The local part is a single-node file system called Episode, which is similar to a standard UNIX file system on a stand-alone machine. The wide-area part is to collect all these individual file systems together to form a wide-area file system.

The DFS in DCE is just like an application, and uses all facilities of DCE. The DCE threads provides the ability to allow users access multiple files simultaneously, RPC offers a bridge for communication between clients and servers, Distributed Time Service (DTS) synchronizes server

clock, the directory service allows file servers to be located, and the security server protects the files by unauthorized accesses.

Each file and directory in DFS is protected by ACLs, which contain a list of entries. Beside the read, write, and execute operations can be assigned to users analogous to the standard UNIX file systems, ACL also allows insert, delete and control. The insert and delete operations are for directories, and control operation is for I/O devices subject to the IOCTL system call [23].

DCE's DFS also supports data replications, load balancing and fault tolerance [17, 23]. The replication server keeps tracks of all replicas of filesets up to date. There is only one master copy of the data that is allowed to be written and read, and one or more replicas for read only. If one replica is changed by a user or process, the replication server will detect the difference by scanning all replicas. The fileset server manages all filesets in the DCE system. If one disk partition is fully loaded by filesets, while other disks still have plenty of space, the fileset server will move some filesets from disk to disk to balance the load. The overseer server is to make sure that all other machines are still alive.

2.2.5 Conclusions of DCE

At the early stage of design, the main purpose of DCE is to provide a robust distributed computing environment, such that the DCE wasn't designed in any object-oriented approach. The tendency of using object technology has forced DCE toward distributed object technology. The latest DCE 1.22 leads to interoperability of different object strategies because DCE provides a foundation for distributed, object-oriented computing without precluding use of other approaches such as CORBA [16]. Thus, we learn advanced distributed computing technologies in DCE.

2.3 Parallel Virtual Machine

Parallel Virtual Machine (PVM) is the mainstay of the heterogeneous network computing research project [7], a collaborative venture between Oak Ridge National Laboratory, the University of Tennessee, Emory University, and Carnegie Mellon University. The PVM project began in 1989 at the Oak Ridge National Laboratory. The main design for PVM is to link computing resources and provide users with a parallel platform for executing their applications, irrespective of the number of different workstations they use and where the workstations are located.

PVM uses the message-passing model to allow programmers to exploit distributed computing across a wide variety of computer types, which include most Unix systems and PCs. The PVM system supports heterogeneity in terms of machines, networks, and applications [7]. With this feature a large parallel virtual machine is possible to be built by using PVM system. PVM consists of two parts: a daemon process on each host (*pvmd*) and a set of library routines (*libpvm*). The usual way for two user processes on different hosts to communicate with each other is via their local daemons. The logical view of an application running on PVM is shown in Figure 2.8.

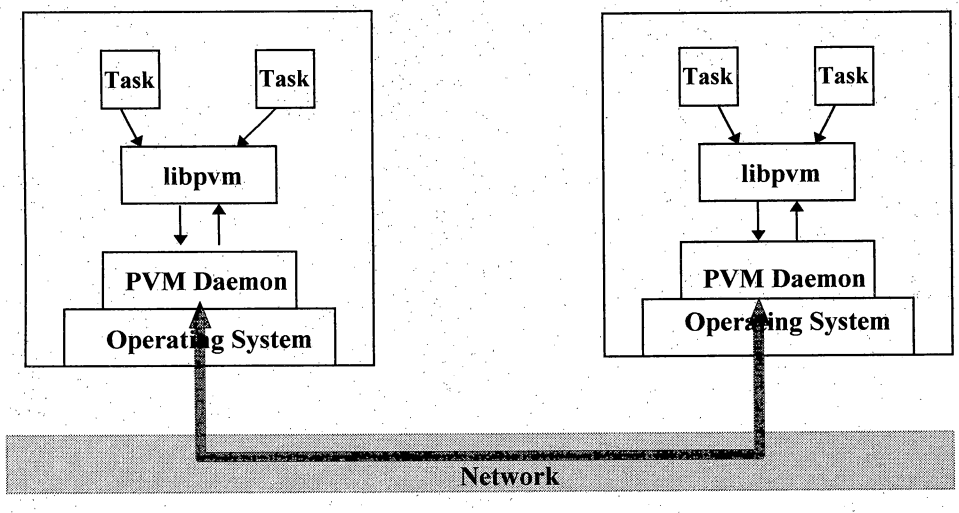

Figure 2.8 The logical view of an application running on PVM.

The following sections will discuss some internal designs and mechanisms used in the PVM system.

2.3.1 Communication in PVM

When a virtual machine is created by PVM, all interconnected machines need to exchange messages over the network for their communication. Since there is no global shared memory in a virtual machine, the message-passing model is adopted by PVM.

In order to support heterogeneity in PVM, the message-passing model needs to be built by using standard communication protocols. PVM uses Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) for the intercommunication, and Unix-Domain sockets mechanism is used to apply on TCP and UDP. Because majority of platforms for PVM are Unix systems, PVM adopts the most prevalent application program interface (API), Berkeley sockets, to be the communication medium for daemons and tasks.

TCP provides a connection-oriented communication service on Internet Protocol (IP). By using TCP, messages can be delivered reliably, but there is overhead to build TCP connections if the need of connections is large. UDP is a connectionless transport protocol which does not require two hosts to set up specific connection route before sending message. However, the messages delivered by UDP are unreliable because the sender can send messages without the

receiver's acceptance. Thus, the acknowledgment and retry mechanism can improve the reliability of UDP.

To make efficient communication in PVM, TCP and UDP are used in different categories:

a) Pvmd-Pvmd. The communication within pvmds are across the network. Thus, to avoid the network traffic and overhead of setting up connections, daemons communicates with one another through UDP sockets.

b) Pvmd-Task and Task-Task. Since tasks perform computing and I/O, they can not be interrupted. If UDP is used for communication, a task might be interrupted to give a retry for a lost packet during computing. Thus, TCP sockets are used in Pvmd-Task and Task-Task.

To provide fault tolerance, a pvmd will recover from the loss of any foreign pvmd by preparing a copy of task before sending to the other pvmd. But the PVM will not migrate the executed task to another host, since it does not have checkpoint.

2.3.2 Conclusions of PVM

The PVM system is not a complete distributed system in general. The goal of PVM is to offer a distributed and heterogeneous network computing environment to users to be

able to solve large computations. Users can collect all available machines to be his/her virtual machine and make use of this huge computing power to solve problems efficiently. Indeed, PVM achieves the heterogeneity in terms of machines (about 40 different Unix machines [7]), networks, and applications.

PVM research group continues to do development of new features and improving the functionalities for PVM. The main milestone for the next version of PVM is to have the capability of collecting Unix systems, Microsoft Windows 95, and Windows NT machines. Users will have the chance to experience the hybrid computing power of MPPs and Pentium.

More and more research groups are cooperating with the PVM groups to study distributed computation. Heterogeneous computing is one of the goals for the Spider Project; therefore, PVM is chosen as an example for our study.

2.4 Condor Scheduling System

The Condor scheduling system [13, 24], developed at the University of Wisconsin-Madison, is designed for a workstation environment so that users can make full utilization of any available workstation for their processing needs. In general, the resources of networks of workstations are under utilized or often idle. When users

face the problem that the ability of their workstation is too small to meet their application needs, the Condor system can schedule users' jobs at idle workstations. Figure 2.9 illustrates the Condor scheduling structure [13]. Each workstation has a local scheduler and a background job queue. One workstation holds the central coordinator in addition to a local scheduler and a background job queue. The central coordinator polls the workstations to know the status for each station (available to serve and background jobs waiting).

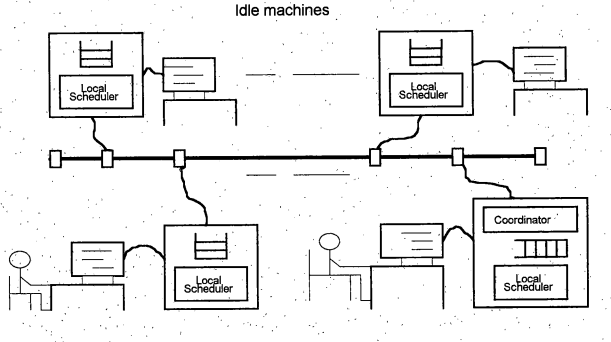

Figure 2.9 The Condor Scheduling Structure.

The main advantage of Condor is fault tolerance. When a user's program is migrated to an idle machine by Condor, the Condor central manager will checkpoint the program periodically. If the machine is running the migrated Condor user program and another user sits down to use this machine, Condor will terminate the Condor user job and save all the

information of running job to a checkpoint file. Then, the Condor central manager will look for another idle machine and transfers the job there. The job will restart by reading the checkpoint file and manipulating its state so as to emulate as accuracy as possible the state of the previous job at checkpoint time. Thus, the Condor user job does not need to restart at the beginning. This scheme also applies to the situation when a machine fails.

The checkpointing of a program in Condor scheduling system is the saving of the state of the program. In order to restart the program and continue the execution on another machine, Condor saves the state of the program including: text, any initialized and uninitialized data, stack areas, the status of open files and file descriptors, and any special handling requested for various signals [13, 24]. Thus, the removed program can be restarted on another idle machine without losing any information of program execution.

2.4.1 Conclusions of Condor

The Condor scheduling system is able to give us a good example on how to provide a fault-tolerant computing environment. The Spider System should be trusted by users to execute their jobs and expect their jobs to terminate normally.

CHAPTER 3 The Spider System

This chapter will discuss the specification and design of architecture and functionalities of the Spider system. The main design goal for the Spider system is to design a distributed system in object-oriented approach, in order to provide a flexible and easily extensible testbed environment for research on distributed and parallel computing in the Department of Computer Science, California State University, San Bernardino.

3.1 Goals for the Spider System

The Spider distributed system will include all the various available machines in our department. Before designing the Spider system, we need to specify the system's goals that are applicable to our department. In general, the Spider system will achieve the following specific goals: portability, heterogeneity, transparency, flexibility, extensibility, and fault tolerance.

3.1.1 Portability

The labs in our department include different kinds of workstations, which run on several implementations of UNIX operating systems (OS) and Microsoft Windows 95. The Spider

system will need to run on different hardware platforms and operating systems. To conquer and achieve the portability for different machine architectures, the Spider system will need a microkernel OS, which will minimize the size of basic system. Thus, the system can be easily ported to other machines by reconfiguration.

3.1.2 Heterogeneity

In a distributed system, heterogeneity is one of the main characteristics. Different architectures of machines (RISC, supercomputers, IBM compatible PC, etc.), operating systems (UNIX and Windows 95), and network protocols are possible hindrance in the construction of a distributed system. For example, the PC machines use the little endian byte sequencing of integers, and the Sun SPARC uses the big endian format. Any data transmitted between these two machines without conversion will definitely cause an incorrect data received by one machine. The Spider system will have the ability to handle all different data formats transmitted between different platforms. Thus, users can make full advantage of all resources in a heterogeneous environment in our department.

3.1.3 Transparency

The main concept of transparency is to hide the distribution of resources from the users. The users are not aware of where the resources are located, how the program is executed by parallel or distributed manner, and how the requested object is implemented. Tanenbaum [23] classified transparency into five aspects of a distributed system, as shown in Figure 3.1.

Kind	Meaning
Location transparency	The users cannot tell where resources are located
Migration transparency	Resources can move at will without changing their names
Replication transparency	The users cannot tell how many copies exist
Concurrency transparency	Multiple users can share resources automatically
Parallelism transparency	Activities can happen in parallel without users knowing

Figure 3.1 Different kinds of transparency in a distributed system.

The Spider system's different functionalities will need to have different kinds of transparency. These functionalities will be discussed in Section 3.3 which are: distributed file system (replication and location transparency), scheduling service (migration transparency), clock synchronization (concurrency transparency), and distributed computation (parallelism transparency).

3.1.4 Flexibility

Users can easily plug in their research implementations into the system and test their algorithms. Furthermore, The Spider system will be able to provide an efficient way for users to submit their jobs. For example, when a user wants to execute a computation, the Spider system can migrate this job to an idle machine to do the computation. If the Silicon Graphics Power Challenge machine is available, the Spider system will set it as a primary choice to achieve a better computing performance. By providing a flexible and intelligent system, users can focus on their research and results from the system.

3.1.5 Extensibility

The Spider system will not only provide flexibility, but also extensibility. The object orientation is able to support a distributed system to be extensible. Any new object (functionality) can be added onto the system to extend the system's functionalities. Also when there are new machines connected to the Spider system, the Spider will have the ability to scale up by including these new machines.

3.1.6 Fault Tolerance

In a distributed system environment, it is possible that one machine may crash or fail due to a fault in some components, such as processor, I/O device, cable, or software [23]. In such a condition, the users are not necessarily informed of the event, but the Spider system needs to provide a recovery algorithm to rebuild the ongoing activities on the crashed machine. By achieving fault tolerance, the system can minimize the loss of information and be more reliable to execute users' jobs.

3.1.7 The Limitation of the Spider System

The Spider distributed system is designed to be built within the CSnet (Computer Science Network at CSUSB), which includes the local area network (LAN), ATM backbone network, and Windows NT network (see Figure 3.2). However, the design of the Spider System should be able to apply in the metropolitan area network (MAN) and the wide area network (WAN) in the future. The specification and design of the Spider System will only concern the computing environment at CSUSB, which will cover the current available machines. This thesis will only implement one of the functionalities defined, and that is the distributed computing functionality of Spider.

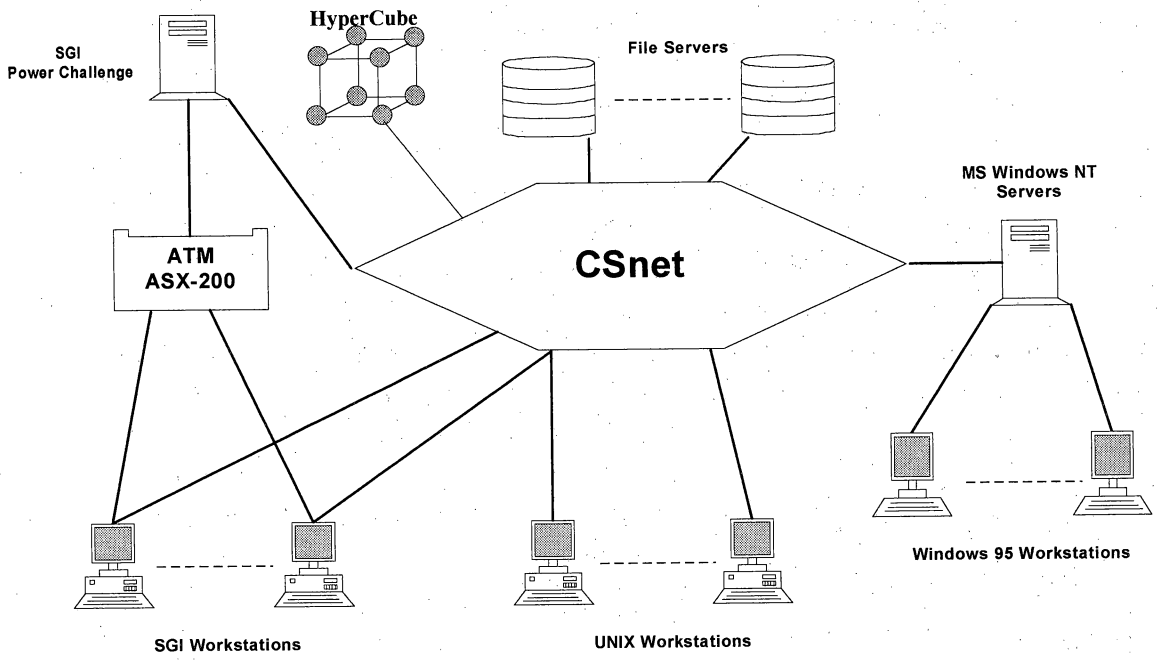

Figure 3.2 The overview of the CS network at CSUSB

3.2 An Overview of the Spider System

Due to the fact that different machines running on various Unix operating systems in our department, the first prototype of the Spider system will be built on top of current Unix OS to achieve portability. Figure 3.3 depicts the architectural overview of Spider and its corresponding equivalence to the OSI model (Open Systems Interconnection Reference Model).

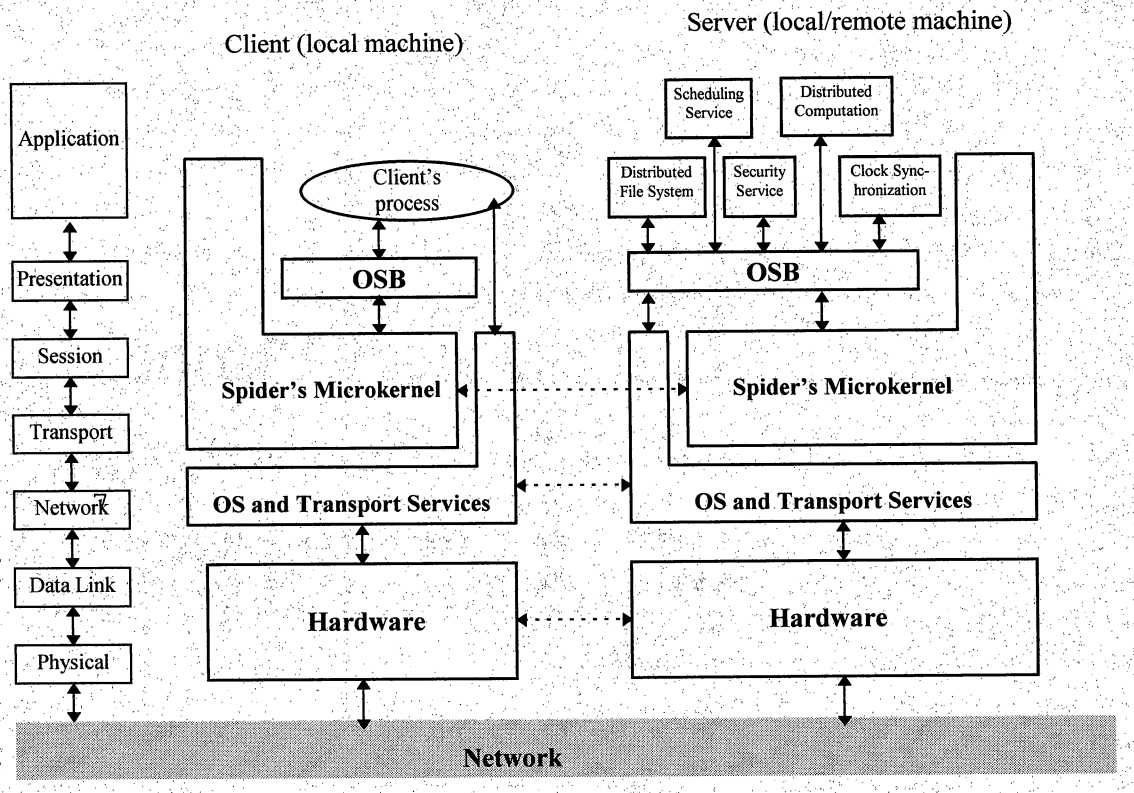

Figure 3.3 The architectural overview of the Spider system in OSI model.

Since we are using object-oriented approach (OOA), we need to treat each service as an object. In OOA, we can provide a testbed environment for researchers to be able to test their work by replacing existing objects and adding on their new objects.

Examples of objects in the Spider system are: distributed file system, security service, clock synchronization, scheduling service, and distributed computation service.

In Figure 3.3, the basic architecture of the Spider system consists of a set of clients (processes), a set of services, an Object Service Broker (OSB), and a Microkernel. Clients may use the OSB via Microkernel to access remote/local services, or directly use any existing OS services. OSB responds by contacting providers of a service and establishing a connection between the client and the service provider. The provider of a service is referred to in the Spider system as a Functionality Server (FS). A service can be defined by an object-oriented interface and implemented in C++. For example, a simple FS may only provide a basic service, such as sorting arrays. A complex FS may provide a more sophisticated service such as file system.

Each FS is an object or provides a group of objects to offer service that is encapsulated and hidden from the clients. The clients are only interested in the behavior of a service, not in its internals. Thus, clients don't know what kind of sorting algorithm is implemented in the FS.

It is important to support the idempotency in a distributed system. The Spider System needs to support idempotent operations to eliminate any redundant request. For a sequence of same requests from a user, Spider must treat these requests as a single request and provides the

same result to the user. A transaction number can be assigned to a user request, in order for Spider to determine that the user's request is repeated or not. The idempotency can reduce any unnecessary or redundant access to the Spider system.

The following sections will discuss the components of the Spider system in more detail.

3.2.1 Microkernel

The Spider distributed system will be a kernel-based operating system. This system will be structured as a collection of cooperating servers running on top of a minimal kernel. Structuring systems in this manner offers a number of potential benefits including ease of distribution, reconfigurability, extensibility, portability, protection and correctness [6, 8]. Like other microkernel operating systems (e.g., the Spring's nucleus [9], and the Mach's microkernel [1]), the Spider's microkernel has four primary abstractions:

a) Process and thread management

The processes are the same as processes in the Unix systems. Each process has an address space. A process consists of one or more threads. All the threads in a process share the address space and execute in a

timeshared manner on a single CPU machine. On a multiprocessor, several threads can be active at the same time.

b) Memory management

The microkernel will provide memory management to handle the allocation and deallocation of memory, paging, and swapping.

c) Inter-process communication (IPC)

The Spider microkernel will provide facilities for the IPC, including pipes, sockets, RPC (Remote Procedure Call), and shared memory.

d) I/O management

All low-level I/O is handled by the kernel. We want to minimize the size of the kernel, so other basic services will be implemented in user level.

Since constructing a microkernel depends on the machine's hardware architecture, it causes its complexity to increase when implementation is from scratch. The Spider's microkernel will have to deal with three major architectures (SGI, Sun SPARC, and Data General) in our CSUSB Laboratories. In the future, the Spider's microkernel will run on every machine, and those Unix services will be moved up to the service objects level.

3.2.2 Communication in Spider

The Spider system will be based on the client/server model. The basic communication paradigms will be the remote procedure call (RPC) mechanism [3] and the BSD socket.

3.2.2.1 Remote Procedure Call

The RPC mechanism provides transparency and efficiency for user programs and applications in the distributed systems. RPC hides conversion of data representation, the address of the remote machine, and communication and system failure from programmers [23].

An RPC system provides the External Data Representation (XDR) library and defines the RPC message-passing protocol. The XDR in RPC system is able to hide all IPC details and offer the ease of developing networked applications. The RPC model provides two communication interfaces, client stub and server stub, which are generated by the RPC compiler. The stubs use the RPC protocol to construct and exchange messages between client and server. The client stub, a dummy procedure, is an intermediate for the client to call the particular functions, where the actual implementations of the functions are on the server side. The client stub delivers the client's request to the server stub and waits for the reply. The server stub unpacks the messages and

invokes the real procedures, then packs and sends the result back to the client stub. The client then receives the result from the client stub as the execution is performed locally. Figure 3.4 describes a basic RPC model.

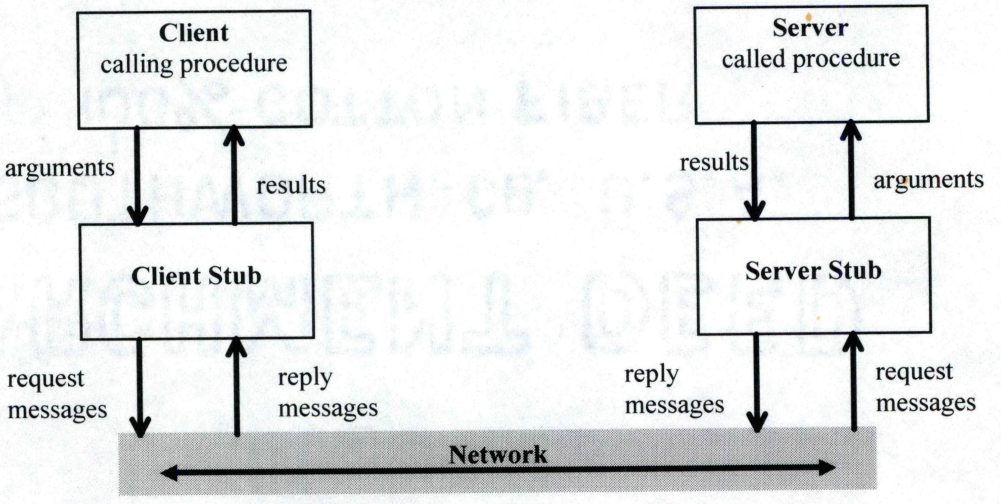

Figure 3.4 Remote Procedure Call.

The communication facility used by the RPC model is provided by the underlying network to deliver messages to the remote machines. The Spider system will use both TCP and UDP protocols for the RPC communication model. To obtain maximum throughput for a bulk data transfer, the TCP protocol can provide a reliable delivery and buffers messages to immediately return control to the user. The buffered messages are flushed when the buffer is full. Thus, the RPC model can be asynchronous and users can make more calls. For low-latency calls, the UDP protocol can be

used and the buffer is flushed immediately, but the user can be blocked. Figure 3.5 shows the comparison of TCP and UDP protocols.

	Protocol Type	Data Type	Transmission
TCP	Connection -oriented	Stream	reliable
UDP	Connectionless	Datagram	unreliable

Figure 3.5 Comparison of TCP and UDP protocols.

To use a remote procedure call in the Spider system, the communication protocols can be system-defined or user-defined depending on the application. The structure of the Spider system is based on the RPC-like model to provide a transparent and object-oriented distributed computing environment.

3.2.2.2 Berkeley Sockets

To use the Berkeley Sockets interface, programmers need to handle all the details of communication in the programs. Although using socket interface will be more difficult for programmers to debug and produce programs, programmers can have more control on the data transmission.

The basic concept of the socket communication is based on establishing a phone-like line between client and server programs. Once the connection has been established, the client and server can exchange information through this particular line.

Since the socket interface can directly talk to the network without any other interface, it is much easier for programmers to use this mechanism to provide more efficient services. In the Spider system, some functionalities will need to provide broadcasting message ability, which will be implemented using socket interface. Furthermore, for a cross-platform connection, the RPC mechanism may have difficulty to apply on different platforms because most RPC compilers are system-dependent.

In Chapter 4, the socket interface will be used for implementing the distributed computation for the Spider system.

3.2.3 Object Service Broker

Object service broker (OSB) is the central component of the Spider system, handling the communication between all objects in the system, regardless of their location, platform or implementation. The main idea of OSB is similar to the subcontract of the Spring operating system [10], the

DCE's Cell Directory Server [17] and OMG Object Request Broker (ORB) [16].

OSB manages the interaction between client and server objects. This includes "marshaling" and "unmarshaling" of requested parameters and results. The marshaling operation of OSB is to transform the request and object into marshaled form and send to server. The unmarshaling operation is to receive the incoming object from the server, extract information from the object, and send result back to the client. Figure 3.6 depicts the client requesting server object through OSB.

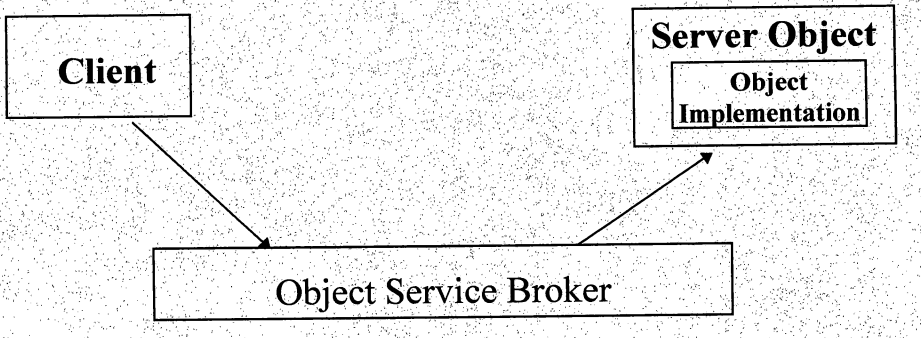

Figure 3.6 Invoking a method on a server-based object

When a client wishes to perform an operation on a server object, OSB is responsible to find the object implementation for the request by contacting the Registry Server, to prepare a server object to receive the request, and to make connection for client and server [16].

The invoking operations on server objects can be performed in a static way, which works very much like Remote Procedure Call (RPC). An object defines its interface using Interface Definition Language (IDL) [16, 17]. The IDL definition is then compiled to produce a client stub and a server skeleton, code that typically gets linked into the client and server objects, respectively. The client stub and server skeleton are part of the OSB (see Figure 3.7). To invoke a method in the server object, a client calls a function, a request that, via OSB, is conveyed to and executed in the destination object. At the same time, the client is blocked until the function returns.

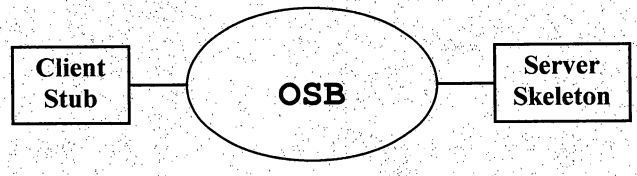

Figure 3.7 The structure of OSB.

The implementation of a simple OSB will be discussed in Chapter 4.

3.2.4 Functionality Server

Functionality servers can be simple or complex, but provide only one service. Each functionality server is implemented as an object and can be accessed via the object-

oriented interface. Thus, these functionality servers can be used as distributed objects and provide for their transparent interaction within a distributed system.

The object-oriented interface used by functionality servers and clients is described in IDL [16, 17]. IDL provides a standard, language-neutral means of describing the public interface of an object. Figure 3.8 describes a simple IDL interface from the specification of OMG's IDL [16].

```
[uuid(70ff8220-6e1a-11cc-89ee-08002b2a1bca)]
interface Sort {
      void sorting ( [in, out] any array, [in] short flag);
}
```

Figure 3.8 A Sort Interface

All data types used in an IDL interface definition (an operation's return types and parameter types) must be either the IDL basic types (**short**, **long**, **float**, **boolean**…), IDL template types (**sequence**, **strings**, **arrays**), or IDL constructed types (**structs**, **unions**, **enums**). In Figure 3.8, the sorting operation takes a pass-by-reference **any** and a pass-by-value **short** to sort any kind of array and does not return a value, but the sorted array will be passed back to the client. The **in** indicates that the parameter is passed from client to server, and **out** indicates that the parameter is passed from server to client.

Since every object must be unique to the whole distributed system, each interface can be assigned a universal unique identifier (UUID) [16]. The unique UUID identifies an IDL interface and give information to the OSB for object invocation. Every FS needs to provide an endpoint [17] for clients to make a connection. Thus, each FS must register its service in the registry server, where the OSB will look up the location of the server by contacting the registry server. A registry server is also one of FSs that exists on every machine and maintains an endpoint table of all servers' objects. Figure 3.9 depicts the steps to create a new FS in the Spider system.

There are three basic steps:

- Register: the new FS needs to register its service with the UUID and endpoint to the Registry Server.
- Update table: Registry Server updates its registry table by recording down new FS's UUID and endpoint.
- Broadcast: Registry Server broadcasts the new information to all other Registry Servers.

After the above three steps are done, the new service is available to the users. Users can access this FS at any workstation.

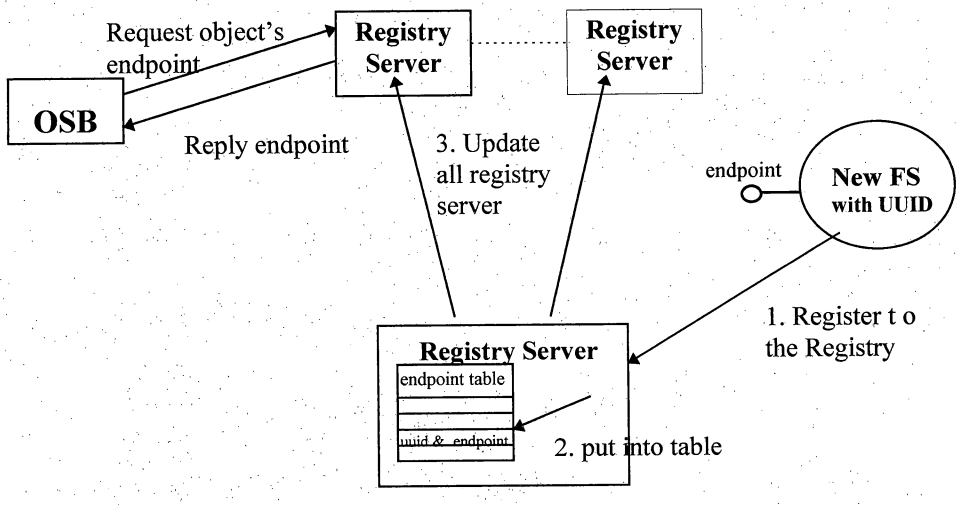

Figure 3.9 New FS registers to the Registry Server.

A single IDL interface may have many implementations, but must contain the operations described by the IDL interface in the public section of the implementation class. The implementation of an object is encapsulated and hidden from the client. Functions on the server side are referred to as skeletons [16, 17], with a skeleton function corresponding to each operation declared in the IDL interface. The skeleton receives invocation requests sent from the client stubs. The skeleton will unmarshal a request and invoke the corresponding member function of the FS. When the member function returns, the skeleton will marshal the return parameters and, with the help of the OSB, send them back to the client side stub.

On the other hand, the member functions declared for the client side class are the same as those declared for the functionality server implementation class. The member functions of the client side class are referred to as stubs. Stubs are invoked by the client program when client sends a request of an operation. They marshal the invocation request and its arguments, send it to the server side with the help of the OSB and then wait for a response. If a response has been received, the stub unmarshals the return parameters and returns to the client program. The inter-process communication necessary for an operation invocation is transparent to the client. Figure 3.10 depicts the interaction of clients and functionality servers. First, the client uses the OSB to request access to services. Second, the OSB responds by contacting FSs of a service and establishing a connection between the client and the FS.

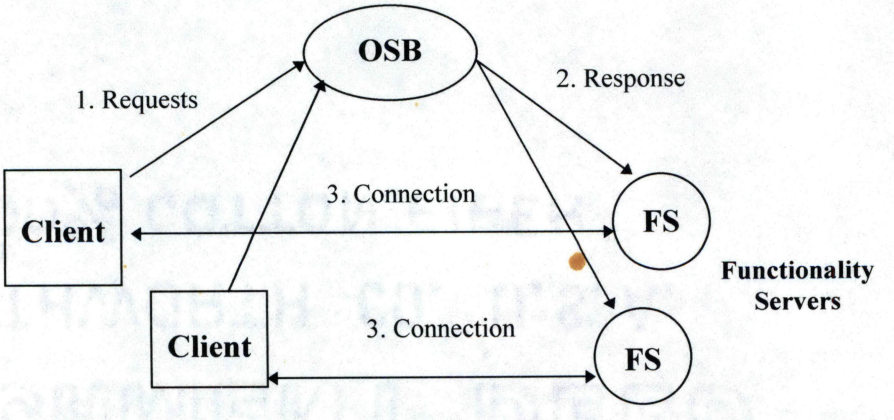

Figure 3.10 The interaction of the clients and Functionality Servers.

3.3 Functionalities of the Spider System

The previous section defines the overview of the Spider system; this section discusses several functionalities which will be developed on the Spider system. The basic mechanisms of each functionality will be defined. These functionalities will include distributed file system, distributed computing, security, and clock synchronization.

Each functionality can be implemented as an object in the Spider system and become a functionality server. Some functionality servers may become clients of other functionality servers. For example, a file object needs to send requests to the security object to get the authentication in order to verify the user's privilege to access the file object.

3.3.1 Distributed File System

The distributed file system (DFS) is an important component of any distributed system. In a distributed system, clients do not need to or should not know that the file system is distributed. Since most of the machines are workstations in our laboratories at CSUSB, files can be stored at any machine.

DFS is one of the functionality servers in the Spider system. In general, this file system uses threads to allow

multiple file accesses simultaneously, RPC and OSB for communication between clients and servers, the security server to protect files, and the clock synchronization server to synchronize server clocks.

Traditional file systems are designed as a central server model, such as Network File Systems [20]. These central server file systems will have performance bottleneck when the system grows large. Examples of file systems, such as xFS [2] and the Spring File System [15], are designed in contrast with traditional file systems. These two specifications give a direction to the design of the distributed file system for the Spider System.

The Spider System is an object-oriented and distributed system that is structured around objects. Therefore, a file or a directory is an object in the Spider system. The basic services for the distributed file system will be discussed in the following sections.

3.3.1.1 Directory and Naming Servers

The naming server will provide file transparency to clients. The clients do not need to know where the file is located. Furthermore, a distributed file system not only needs to have location transparency, but also location independence [23], such that, any file can reside at any

machine. However, the location independence is not easy to achieve, but it is a desired property to have in a distributed system.

The directory server is able to provide the creation and deletion of directories, using naming server to name and rename files, and moving location of directories. The global root directory should be viewed the same way in all clients. The directory server maintains the directory path table in every machine, but not for all the directory paths. For example, if a user looks up the path /A/B/C, the user sends a message to the directory server, which will find the location of A. According to the directory path table, the server having A will provide information of B. Then the same for C, the user will be able to get the information of files under C directory [23] (see Figure 3.11).

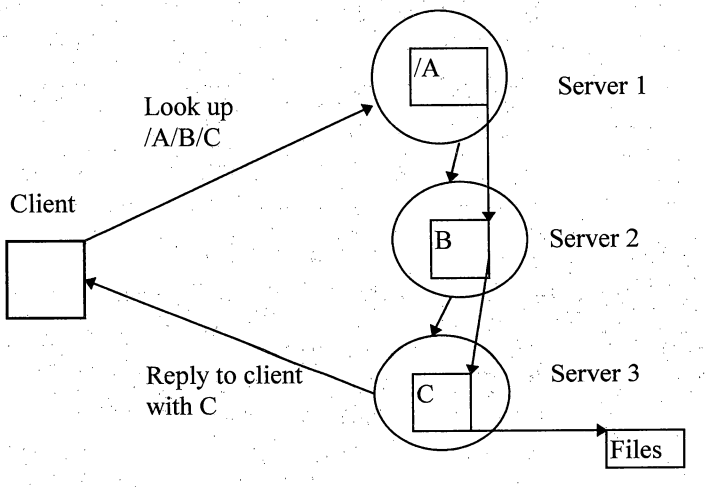

Figure 3.11 Directory lookup.

The naming server allows any file object to be associated with any name. This association of name and object is called a name binding [15]. Each name binding is stored in a directory object, such that every file name is unique in this directory object.

Both directory server and naming server is secured by an access control list (ACL). An ACL can protect every file object and directory object. Only authorized clients can be allowed to access the particular objects in DFS. ACL will be discussed in the security section.

3.3.1.2 Caching Server

The DFS of the Spider system should be stateful [23]. Although a stateless file server tends to have fault tolerance, the systems that keeps less client's state information force clients to communicate with the server more frequently, which will cause more network traffic. In the Spider system, each file server has to keep track of which clients have cached copies of the file objects. However, the caching server of the client side has the duty to report to the file server, if the cache file object has been modified. Because the file server tracks all the cached file objects, it will notify (invalidate) all other clients

holding a copy of the cached file object. This will ensure that all clients will be able to access the latest data.

The main responsibility of the caching server is to keep data consistency of all cached objects in the Spider system. The DFS of the Spider system will use the write-invalidate protocol [21] to keep cache consistency. It will use the single writer and multiple readers semantic. By using file locking, all files are kept consistent. The locking server can be elected by the file object's server initially. Every client must request file lock before reading or writing a file. Because the file object is cached on the client's machine, the original file object must be updated and all other copies are invalidated, once the client modifies the cache file object.

Using the caching server, each object can be replicated and sent to clients. The caching server also needs to interact with the microkernel's virtual memory manager in order to provide an efficient file service.

3.3.2 Security

Security is a major concern in a distributed system. We want to avoid any unauthorized use and access to the Spider system. Thus, the security of the Spider system will consist

of two servers -- authentication server and Access Control List (ACL) server.

3.3.2.1 Authentication Server

In a timesharing system, a user logs on a machine by typing the user's name and password. If a user logs in successfully, the kernel keeps track of the user's identity and permits or refuses access to files and some other resources based on it [23]. In the Spider system, once a user logs in and gets authenticated by the security server, it's not necessary to keep track of the user. Thus, the security service must provide the authentication.

A well known authentication server is Kerberos [22]. The user receives a ticket from the Kerberos authentication server after exchanging encrypted message. The ticket will allow the user to access the network services, which specifies the limitations of user's access, then the ticket can be sent over the network in the distributed system without sending the user's password. The Spider system will use Kerberos as part of the security service.

3.3.2.2 Access Control List Server

The Access Control Lists (ACLs) [14, 21] allow a user to receive from the file server permission to operate on the

particular file objects or directory objects that are stored in the file server. The ACLs give the authorization to clients and inform which users and groups may access the resources. Therefore, each object in the Spider system is protected by an ACL. When a user requests an invocation on an object service, the ACL server will issue an ACL to the object depending on the user's ticket.

There is no central ACL server in the Spider system, but each file object is associated with its ACL server depending on its location. To modify the access rights of the objects, only the object's owner has the ability to perform a hierarchical control [21], which allows the owner to modify the ACL of the object and all the objects below it in the hierarchy.

The basic steps for a user to log in the Spider system and use the system services can be described as follows (see Figure 3.12):

a) Log in the system. The user types the username and password.

b) Get a ticket. The Kerberos authentication server authenticates the user and grants the ticket to the user.

c) Request service. The user sends the request to the OSB.

d) Issue the ACL. The OSB contacts the ACL server to issue a proper ACL to the user.

e) Give ACL to the object. The ACL server assigns the capability for the user to access the object.

f) Reply to the user. The object performs the requested services according to the ACL and gives the results to the user.

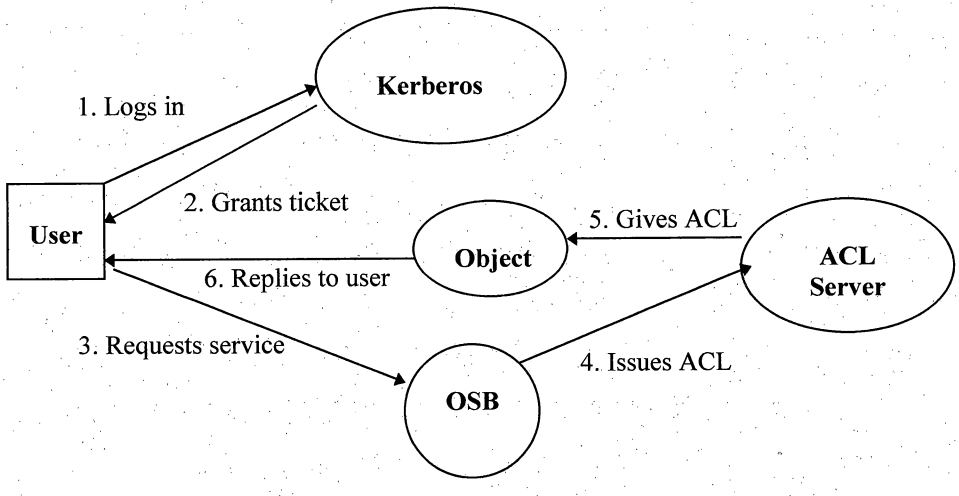

Figure 3.12 The overview of the security model.

The security in Spider should be stateful, but we choose it to be stateless. Spider is designed as a distributed object system, and every server object invoked by a user has an ACL to accompany with it. Because a user receives a ticket from the authentication server and this ticket gives the certain privilege of accessing server

objects, the security in Spider does not need to record all information of user's activities.

3.3.3 Clock Synchronization

In a distributed system, the absence of a global clock [21, 23] may cause the system to be in an inconsistent state because all workstations' clock are not synchronized. There are some algorithms discussed in [21, 23] on how to synchronize clocks in a distributed system. However, most of these algorithms use a single time server to be responsible for the clock synchronization, which is not suited in a distributed system. Park [18] proposed an optimistic concurrency control mechanism based on a clock synchronization, which provides a flexible and efficient way to synchronize the global clock in a distributed system. DuVall's simulated global clock algorithm [5] represented an improvement over Park's clock synchronization. These two algorithms will be adopted by the Spider system.

3.3.3.1 Park's Clock Rate Synchronization Algorithm

In Park's algorithm, the clock rate server (P_{CRS}) is chosen randomly to adjust the clock rate of the requested machine (P_i). During the clock rate synchronization, another coordinator machine (P_C) is chosen randomly to be the

intermediate for these two machines. The algorithm has the limitation that all machines run on the same LAN because the constant message transmission is assumed. The clock rate synchronization is illustrated in Figure 3.13.

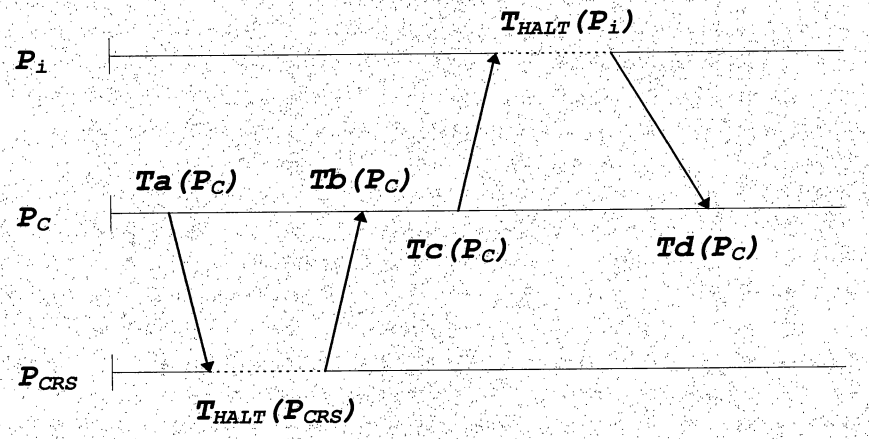

Figure 3.13 Clock Rate Synchronization [18].

The relations between these machines can be determined by the following equations [18]:

$$T_{TRANS}(P_{CRS}, P_C) = [Tb(P_C) - Ta(P_C) - T_{HALT}(P_{CRS})]/2$$

$$T_{TRANS}(P_i, P_C) = [Td(P_C) - Tc(P_C) - T_{HALT}(P_i)]/2$$

$$Clock\ Ratio(P_{CRS}, P_i) = [Tb(P_C) - Ta(P_C)]/[Td(P_C) - Tc(P_C)]$$

Park proposed a clock rate synchronization, where the clock rate server can be chosen randomly. This algorithm is suitable for a real distributed system. The clock synchronization only need to be done once initially. Since the Spider system is designed for the Department of Computer

Science at CSUSB, Park's algorithm is applicable to the Spider system.

3.3.3.2 DuVall's Simulated Global Clock

DuVall's simulated global clock [5] presents a more efficient algorithm for clock synchronization which combines the features of Cristian [4] and Park [18] to be a hybrid clock synchronization algorithm. DuVall refined Park's algorithm to be able to synchronize system clock with the Universal Coordinated Time (UTC) rate, and DuVall's algorithm does not need periodical re-runs for clock synchronization.

DuVall's algorithm adopts the transmission time estimation from Cristian and offset to local time from Park [5]. Periodically, at least with each configuration change, this algorithm will need to be run or re-run. If the coordinator has accessed to the UTC source, the physical global time can be maintained in the system.

In the Spider distributed system, the clock synchronization server will adopt DuVall's algorithm to provide a consistent global clock. The clock synchronization will apply when the system start up. If any machine has crashed and then boots up again, only that particular

machine will perform the clock synchronization, not all machines in the system need to re-synchronize again.

3.3.4 Scheduling Server

The scheduling service aims to maximize the utilization of workstations, so that a user is able to use idle or under utilized workstations. The main goal of a distributed system is that a user can make full use of all available resources, including computing power of workstations. However, we don't want users to interfere with other users' work; so, only idle and under-utilized workstations are available for users in the Spider system.

The Condor scheduling system [24] presented a successful implementation of process scheduling. The Condor system provides a scheduling mechanism to schedule long running background jobs at idle workstations. It also has the checkpoint and migration facilities in order to support the fault tolerance. If an idle machine is logged in by a user or crashed, the migrated job will be stopped and be migrated to another idle machines. That job will restart at the point where the program stopped according to its checkpoint.

Since the checkpoint mechanism is used for the scheduling service, it is necessary for the scheduling

server of Spider to be stateful. The main components of the scheduling service are scheduling manager and pooling server.

3.3.4.1 Scheduling Manager

In the Spider system, the scheduling server will exist on every machine, which is different from a central manager in the Condor system. When a user executes a long running background job, the local scheduling server will call its pooling server to get the information of idle machines. Then a scheduling manager will be elected randomly from the pool to monitor all the scheduling activities for the user's task. The local scheduling server will be the shadow of the scheduling manager to keep the most current state of the task execution. Therefore, if the remote machine crashed or becomes busy, where the scheduling manager and pooling server are running on, the local scheduling server is able to re-elect another idle machine to be the new scheduling manager.

3.3.4.2 Pooling Server

When a user wants the scheduling service, the pooling server will collect all idle and under utilized workstations in its pool. Each machine will be assigned a priority number

according to the machine's performance. For example, the highest priority can be the Power Challenge machine, and the lowest priority can be the PC. This will guarantee to provide users with the most efficient computation for their applications.

During execution, if a user attempts to log on the machine which is executing a migrated job, the Spider system will prompt a message to warn the user. The user may choose another machine, or still log on this machine if the user does not mind to experience a degraded performance. This mechanism will reduce the risk of too much job migration and the complexity of checkpoint.

Condor system is not capable of migrating jobs using a central manager because if the central manager goes down, all the information on migrated jobs will be lost. The scheduling algorithm for the Spider system may involve more messages sent over the network, but it is more suitable and reliable for a distributed system. The scheduling algorithm is illustrated in Figure 3.14.

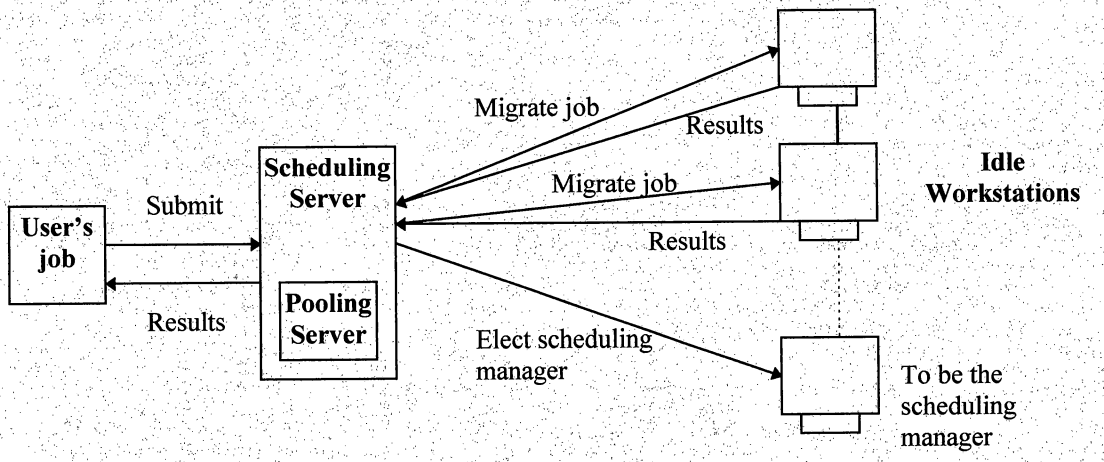

Figure 3.14 Job migration via scheduling server.

3.4 Conclusions of the Spider

In this chapter, the structure of the Spider is specified and designed. To achieve the object orientation for Spider, OSB offers a transparent object invocation for user applications and FSs provide an object-oriented and distributed service environment. Because of the heterogeneous computing environment in CSnet, the Spider's microkernel is needed to implement a suitable infrastructure for the Spider distributed system. The functionalities of the Spider Project are also defined. These functionalities act as FSs in Spider. They are all specified and designed as objects which facilitates ease of continuing the research on implementing the functionalities of Spider.

In Chapter 4, the Distributed Computation Service in Spider is implemented. To prove the validity of the design, OSB, Registry Server, and Task Manager are implemented in object-oriented approach and then tested for correct execution.

CHAPTER 4 Implementation of Distributed Computation Service

The concept of distributed computation is to distribute jobs to several remote machines. Each machine may perform a different function, for example, I/O, problem setup, solution, output, and display. All the machines can also perform the same function, to solve a small part of the data. This is referred to as the SPMD (single-program multiple-data) model of computing [7].

In this thesis, one of the Functionality Server, the Distributed Computation Service (DCS), for the Spider Project is implemented, and only three services: sorting, matrix multiplication, and vector addition are supported by this implementation. The implementation is described in the following sections.

4.1 Implementation of Distributed Computation Service

The implementation of DCS is designed in an object-oriented approach and written in C++ and Java programming language. The Distributed Computation Service has the following components: OSB, Registry Server (RS), Task Manager (TM), and Java Graphical User Interface (GUI).

To design in object-orientation, we must provide objects to have three characteristics [19]: encapsulation, inheritance, and polymorphism.

- *Encapsulation* means that an object's data and methods aren't accessible by the object's users except via its methods. In C++ programming language, we can use private to encapsulate the data and methods from the outside world.
- *Inheritance* offers objects to be reusable. Child objects do not need to implement code, but can instead directly use and build upon the code that is in the parent.
- *Polymorphism* of object-orientation is the most complex to describe. Put simply, polymorphism means that the user of two different objects can, in some ways at least, treat them as if they were the same. For example, two objects, one representing your checking account and another your saving account. These two accounts almost have the same characteristics for your bank activity, both of them offer deposit and withdraw methods. However, the saving account object's withdraw method may probably just check the amount to be withdrawn against the account balance. The transaction either succeeds or

fails which is depending on the requested amount exceeding the balance or not. For the checking account, the requested withdrawn amount may exceed the balance, if the exceeding amount is within the limit of an automatic loan, which can protect against overdrafts.

These important object-orientation characteristics are applied to those objects in this implementation. Figure 4.1 shows the overview of the Distributed Computation Service.

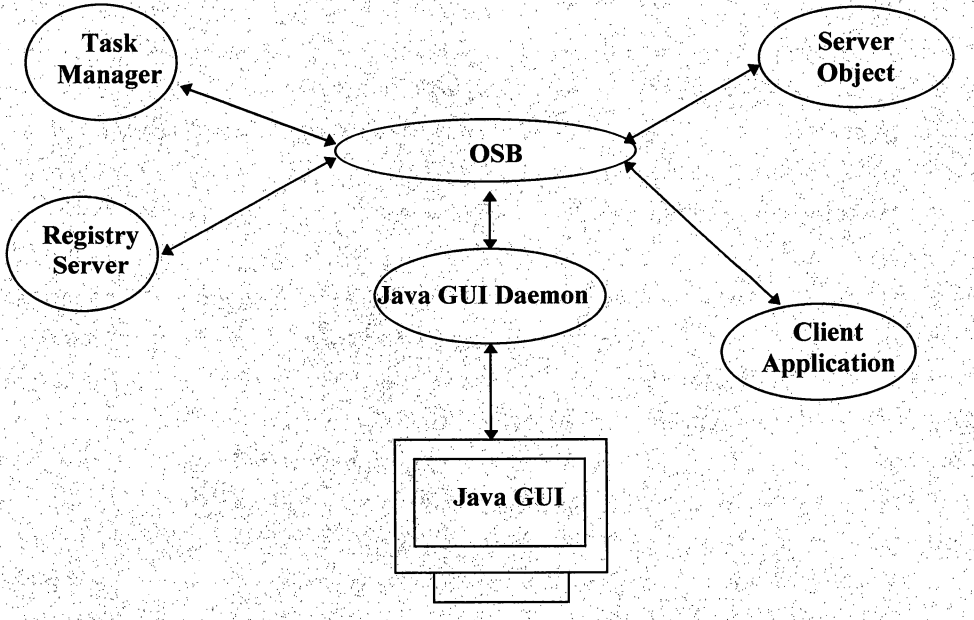

Figure 4.1 Overview of Distributed Computation Service.

4.1.1 Object Service Broker (OSB)

OSB plays an important role in the Spider Project. The main functions of an OSB object are:

- to locate the registered servers from Registry Server (RS) according to the client's request;
- to submit the job to the Task Manager (TM) in order for the TM to monitor the execution;
- to activate the remote server object and convey the client's request to that server object; and
- to notify TM that the job is finished and deactivate the server object.

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

4.1.2 Registry Server

The responsibility of the Registry Server, which manages a service data base, is to provide an available registered servers list by verifying the request from OSB. The assumption for the Distributed Computing Service is that every machine is independent and may or may not share the

same file system with others. Figure 3.9 depicts the actions taken when the new Functionality Server (FS) is registering to the Registry Server and the OSB is requesting servers from the Registry Server.

4.1.3 Task Manager Server

To manage the available servers for each task during computation, TM needs to keep track of all tasks' activities. In a distributed system, we don't want the Task Manager to be centralized and provide fault tolerance for DCS. Thus, a mirror TM object is created and keeps the same information as TM, in order to take over the TM's job when TM goes down. TM and mirror TM are running on different machines and updating data periodically.

OSB has two options for contacting with the TM. First, OSB tries to contact the Task Manager server, if TM is down or no response, OSB will try to contact the mirror TM. If both Task Manager servers are down, then the operation is aborted because the server object can not operate properly without the TM.

4.1.4 Java Graphical User Interface

In order to provide a friendly GUI to users, we choose the Java programming language to implement the GUI for the

DCS. Java applets are able to run on any platforms where there is a web browser (e.g., Netscape and Microsoft Internet Explorer). Therefore, anyone is able to access the Spider's DCS through Internet, if they are authorized users.

Because of the Internet security issue, users can't send files to the original web servers, or open the servers side files via Java applets. Thus, this GUI can only show simple demonstrations to users how the distributed computation is being done. If users want to use these services, they must login in the CSnet and create their own client programs and access the DCS in text mode. Figure 4.2 shows the state diagram of the Java program.

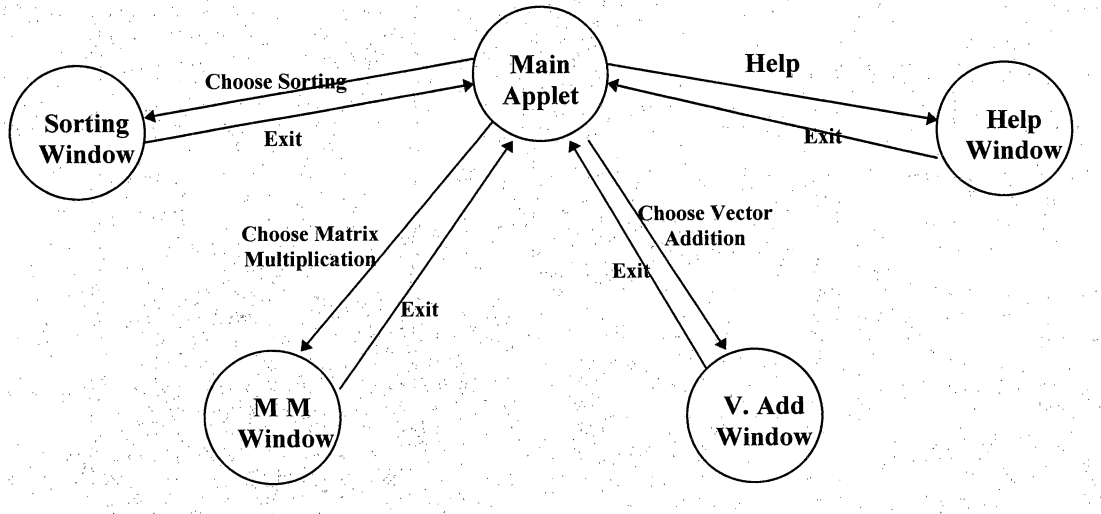

Figure 4.2 State diagram of Java GUI.

Each window (Sorting, Matrix Multiplication, and Vector Addition) uses TCP sockets to communicate to a C++ daemon located in the same machine, where the Java program is downloaded. The daemon provides the most current execution information to the Java applet, and the Java program displays the graphical diagram to users (see Figure 4.3 and Figure 4.4).

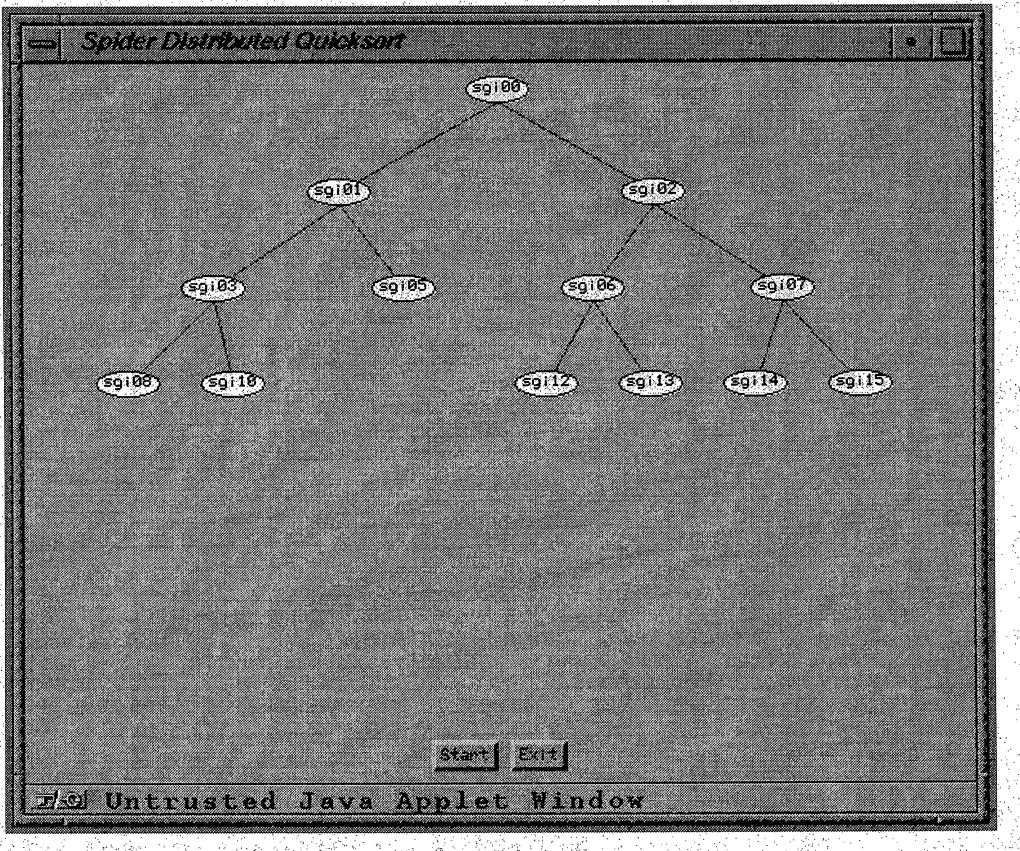

Figure 4.3 A snapshot of the Distributed Quicksort window.

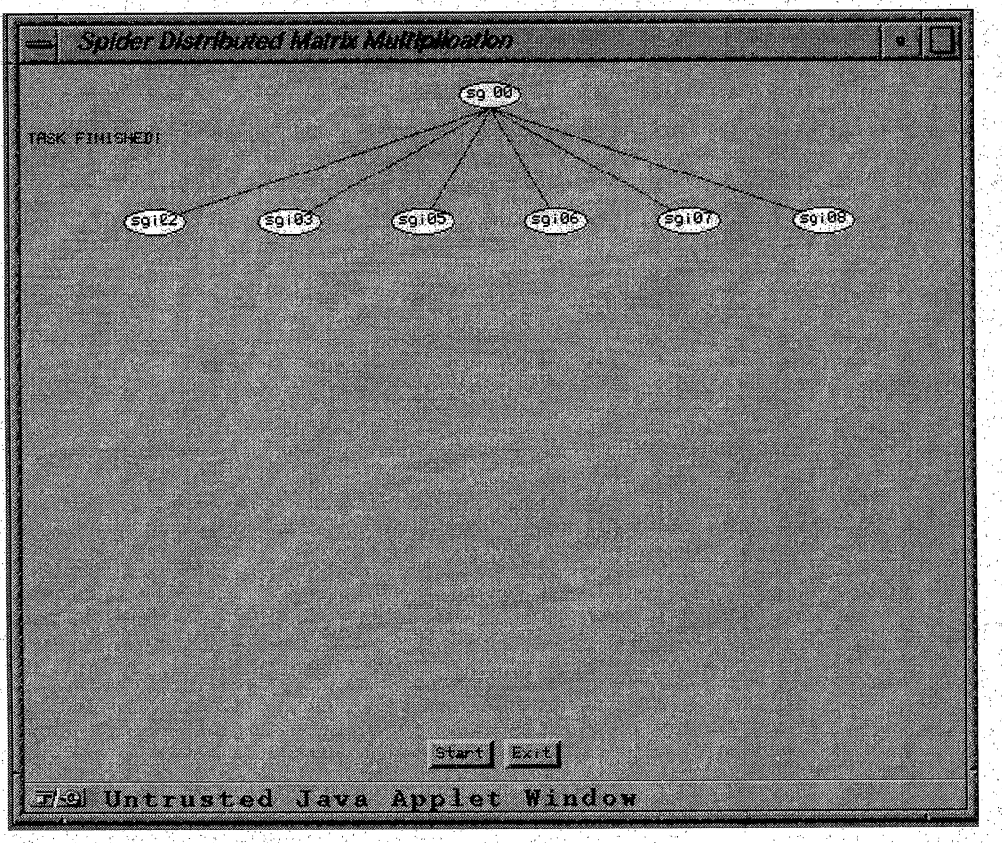

Figure 4.4 A snapshot of the Distributed Matrix Multiplication.

The GUI contains applet, frame, panel, and dialog objects. Since Java already provides hierarchies of common classes, we create these objects by inheriting existing classes. Figure 4.5 shows the hierarchy of classes used for this implementation. The hierarchy shows two categories, one is Java classes library, and the other are user-defined classes. The GUI implementation is based on these objects to display graphical diagrams to users.

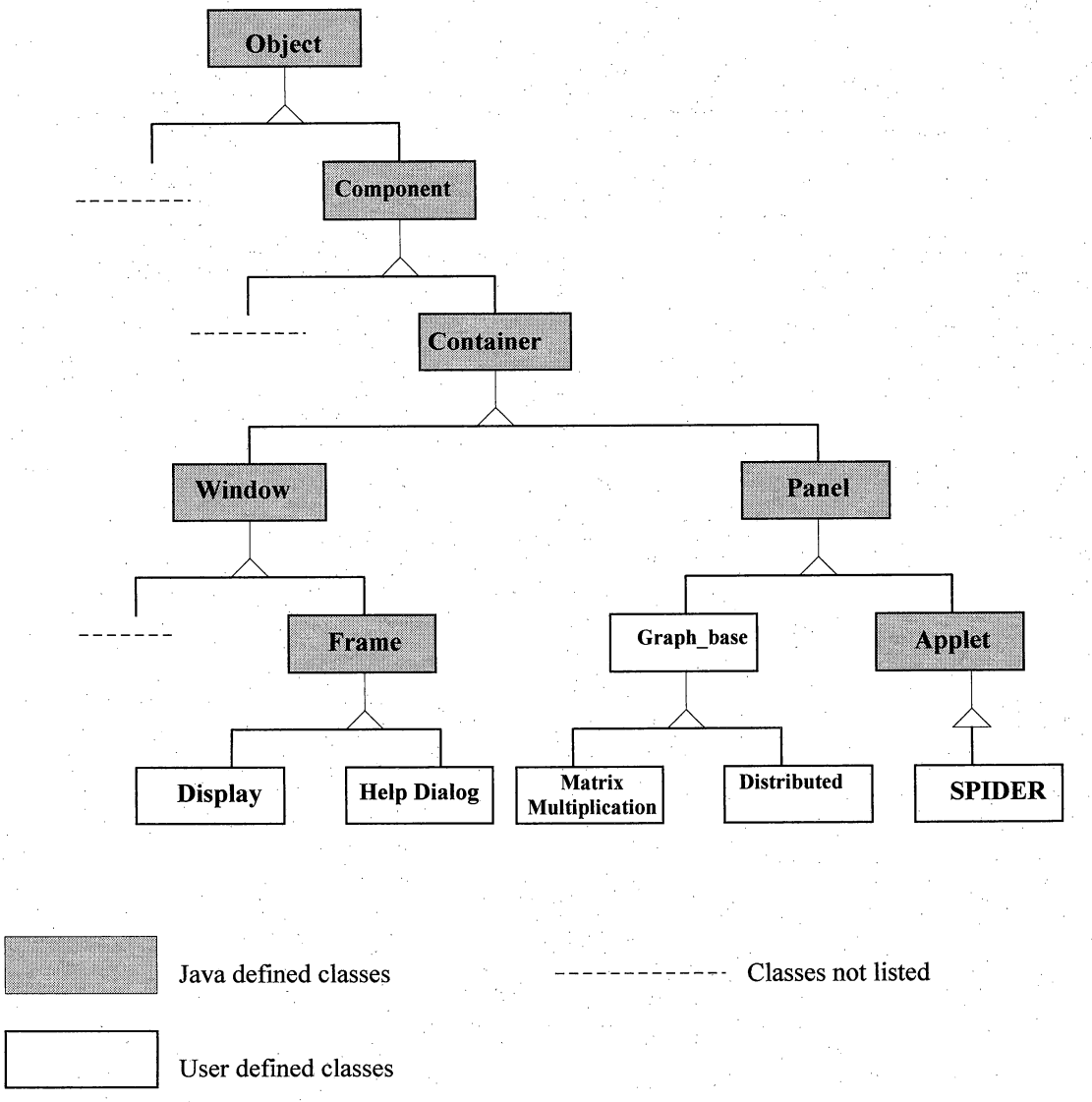

Figure 4.5 The hierarchy of GUI Java classes.

4.1.5 Communication in DCS

RPC and BSD socket interface are used for the implementation's communication mediums. There are three

interfaces that are defined as interface definitions for these service objects: distributed quicksort, matrix multiplication, and vector addition. Because the IDL compiler for the Spider Project is not implemented, the ONC RPC's RPCGEN compiler is used to generate all client and server stubs. These stubs provide the proper data conversions for clients and server objects.

For the implementations of server objects, the BSD socket interface is used for communication. TCP and UDP protocols are employed to suit for different needs in this implementation. The table of Figure 4.6 describes the protocols used for communication in this implementation.

Type	Protocol
client-server	RPC
OSB-Registry	UDP
OSB-Task Manager	TCP
task-Task Manager	TCP
task-task	TCP
SVR_OSB-Task Manager	TCP
task-Java Daemon	UDP

Figure 4.6 The protocol table.

To understand the procedure of the distributed computation in the Spider system, Figure 4.7 illustrates the steps of the distributed computation.

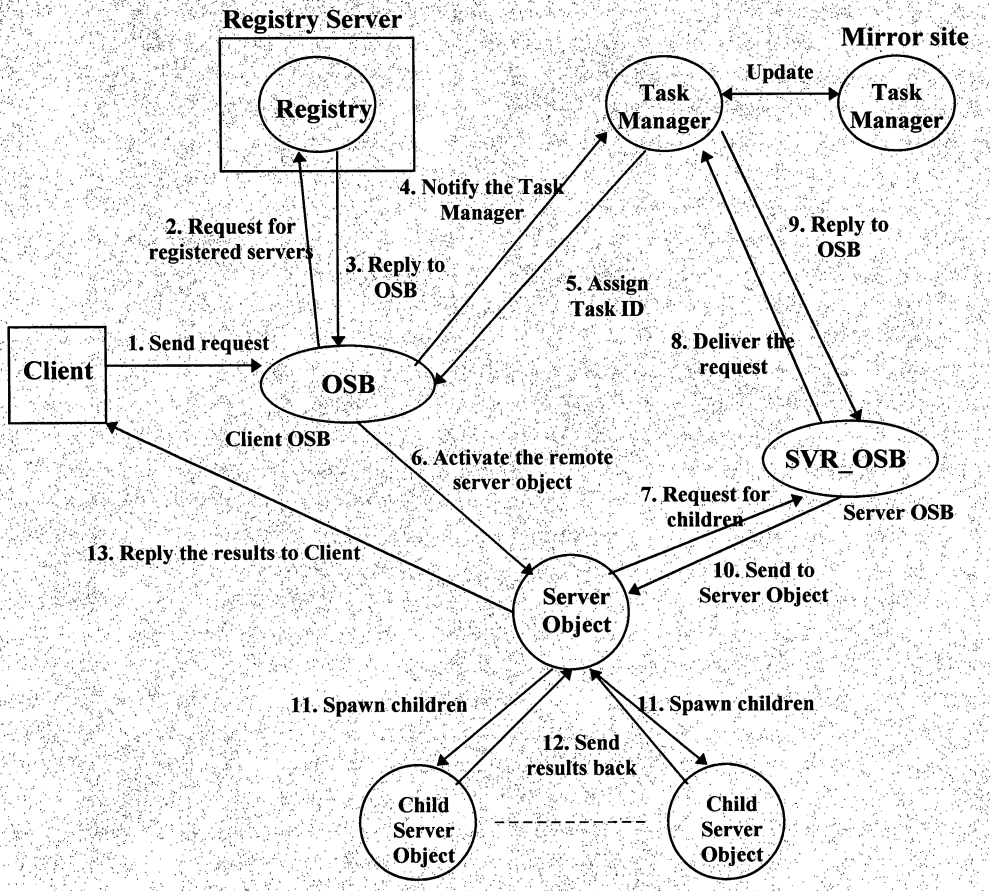

Figure 4.7 The protocols of the Distributed Computation Service.

To illustrate the implementation in object-oriented approach, the object diagram of Distributed Computation Service is shown in Figure 4.8.

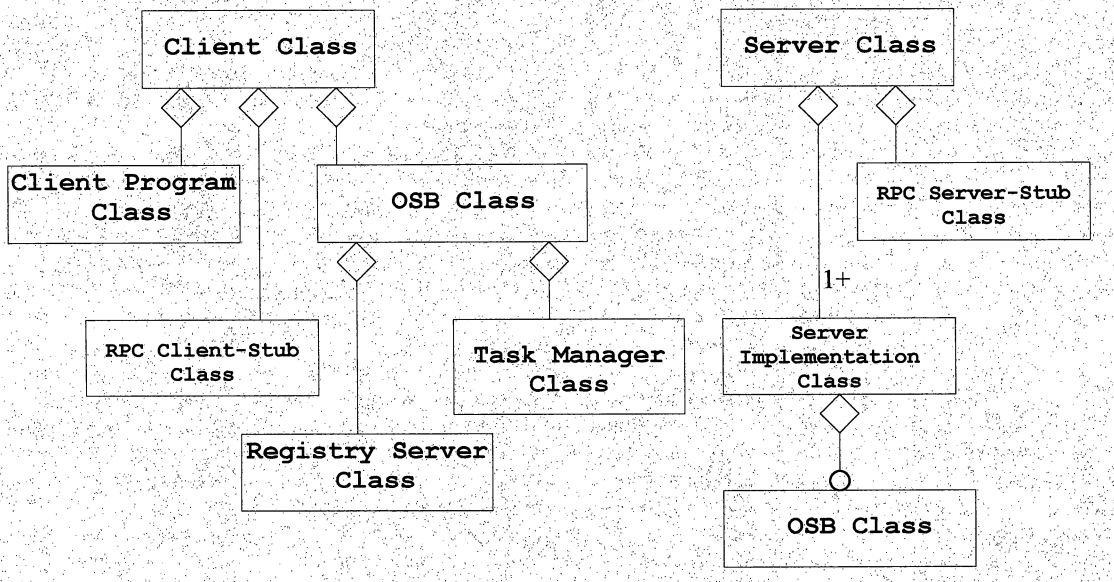

Figure 4.8 Object model of Distributed Computation Service

4.2 Overview of the Distributed Computation Service

DCS is designed to solve large and complex computations in a distributed and parallel manner, which are sometimes difficult to be executed within a single machine. If the computation is done on the user's machine, the user may experience the degradation of performance because most CPU usage is used on this computation. In a distributed computing environment, once the users submit jobs to the distributed system, they can continue doing other work. Users are not aware of knowing how the job is done and where

the job is executed, they just need to receive the correct results.

Sorting, matrix multiplication, and vector addition are common computations. How these three problems solved in a distributed computation is discussed in the following sections.

4.2.1 Distributed Quicksort

The quicksort 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)
/* sort array A from the lo to hi into ascending order on
key P. Key Pm is arbitrarily chosen as the control key.
Position I and j are used to partition the subfile so that
at any time K1 ≤ K, 1 < i and K1 ≥ K, 1 > j. It is assumed
that Km ≤ Khi+1. */
if lo < hi
     then [ i ← lo; j ← hi+1; K ← Km
          loop
               repeat i ← i+1 until Ki ≥ K;
               repeat j ← j-1 until Kj ≥ 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 quicksort is $O(N\log_2 N)$.

The algorithm of distributed quicksort is to define a maximum number of data size that can perform quicksort in one machine. If the size of data is larger than the defined size, then the data is partitioned into two parts and given to two children which in turn apply distributed quicksort recursively. The algorithm is given as follows:

```
Array A[1. . . N], minimal data count is M
procedure Distributed_Qsort(B, lo, hi)
/* B is either part or entire of array A. The lo and hi is
the lowest and highest position of array B, respectively. p
is the pivot point of B after partition. */
if (I have parent)
     then [          /* I am the child */
           receive data from parent
           if ((hi - lo +1) < M)
                then [ call QSORT
                       send the result back to parent ]
           else [
                  partition B into two parts
                  send two parts to two children
                  wait for results from children
                  send results back to parent ]
     ]
else [               /* I am the parent */
           if ((hi - lo +1) ≤ M)
                then [ call QSORT
                       display result ]
           else [
                  partition B to two parts
                  send two parts to two children
                  wait for results from children
                  display result ]
     ]
end Distributed_Qsort
```

The average computing time for the distributed quicksort is still in the form $O(N\log_2 N)$, but N may vary and

depends on the defined minimal data size M. If N is much greater than M, then the resulting sorting tree's depth is increased. That means more children are created during computation, and these leaf nodes will only sort at most $(N/2^{d-1})$ of data, where d is the depth of the binary tree. Thus, the average computing time for distributed quicksort will be $O((N/2^{d-1})\log_2(N/2^{d-1}))$. Figure 4.9 depicts this computation.

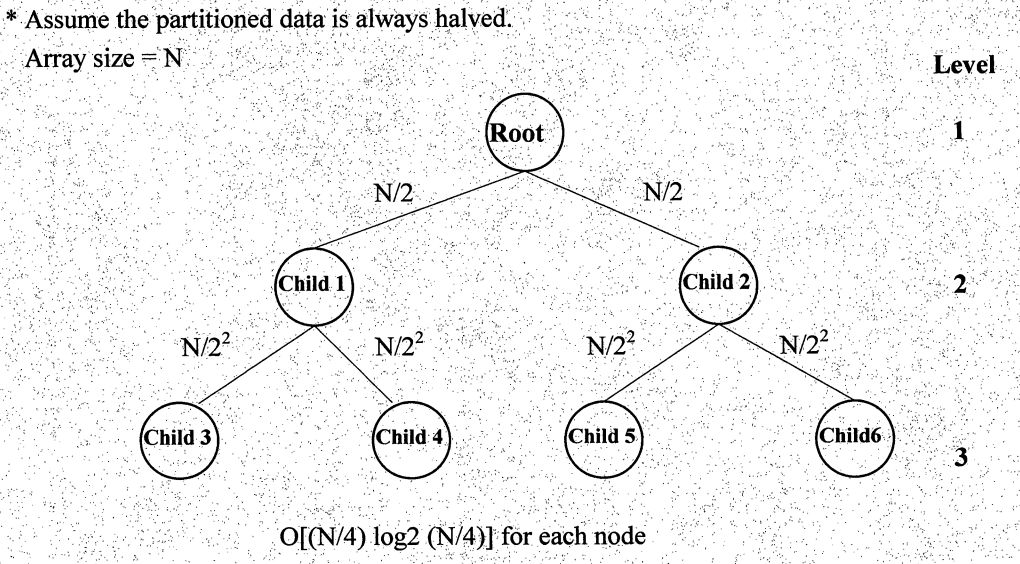

Figure 4.9 Distributed Quicksort tree.

On the other hand, the transmission time may be the main factor that affects the total computing time, and we will discuss this issue in Section 4.3.

4.2.2 Distributed Matrix Multiplication

The average computing time of two NxN matrices is $O(N^3)$, which is relatively higher than quicksort. The algorithm for distributed matrix multiplication is to distribute the data to several machines and every machine performs the computation for the portion of the matrix multiplication in parallel.

Assume that we use M machines to perform the computation. The multiplier is separated into (N/M) sub-matrices, and each child only computes (N^3/M). Therefore, the total average computing time is $O(N^3/M)$. Figure 4.10 depicts the distributed matrix multiplication.

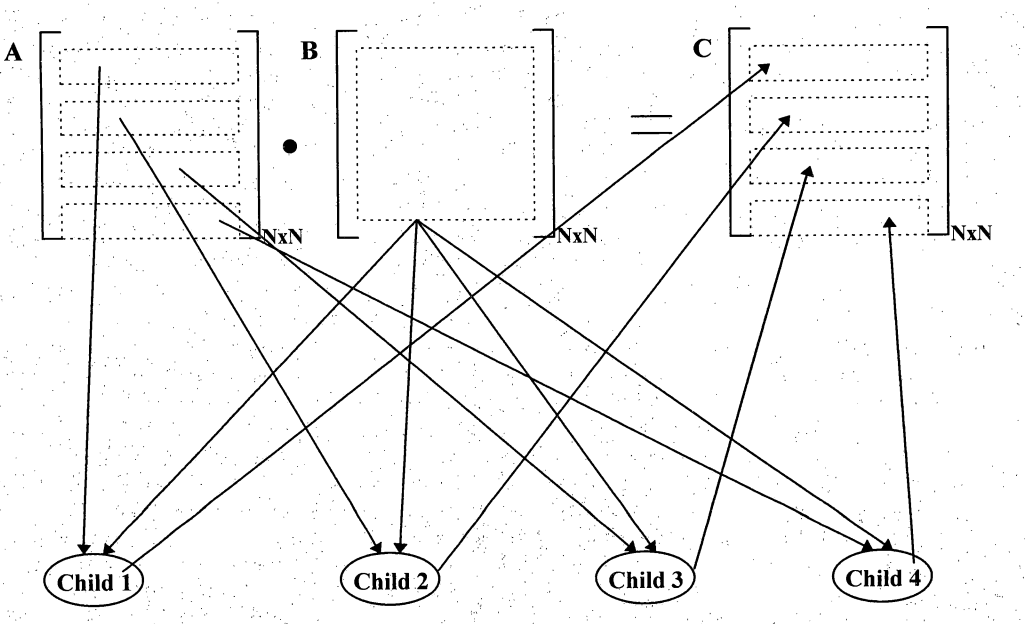

Figure 4.10 Distributed Matrix Multiplication.

Also, we need to consider the transmission time spent in the distributed matrix multiplication which will be discussed in section 4.3.

4.2.3 Distributed Vector Addition

The computing time for the addition of two vectors (arrays) is O(N) where N is the size of the array. If we can fragment the arrays and distribute them to M machines, we can reduce the computing time by computing in parallel. The improved total average computing time will be O(N/M). Figure 4.11 depicts the distributed vector addition.

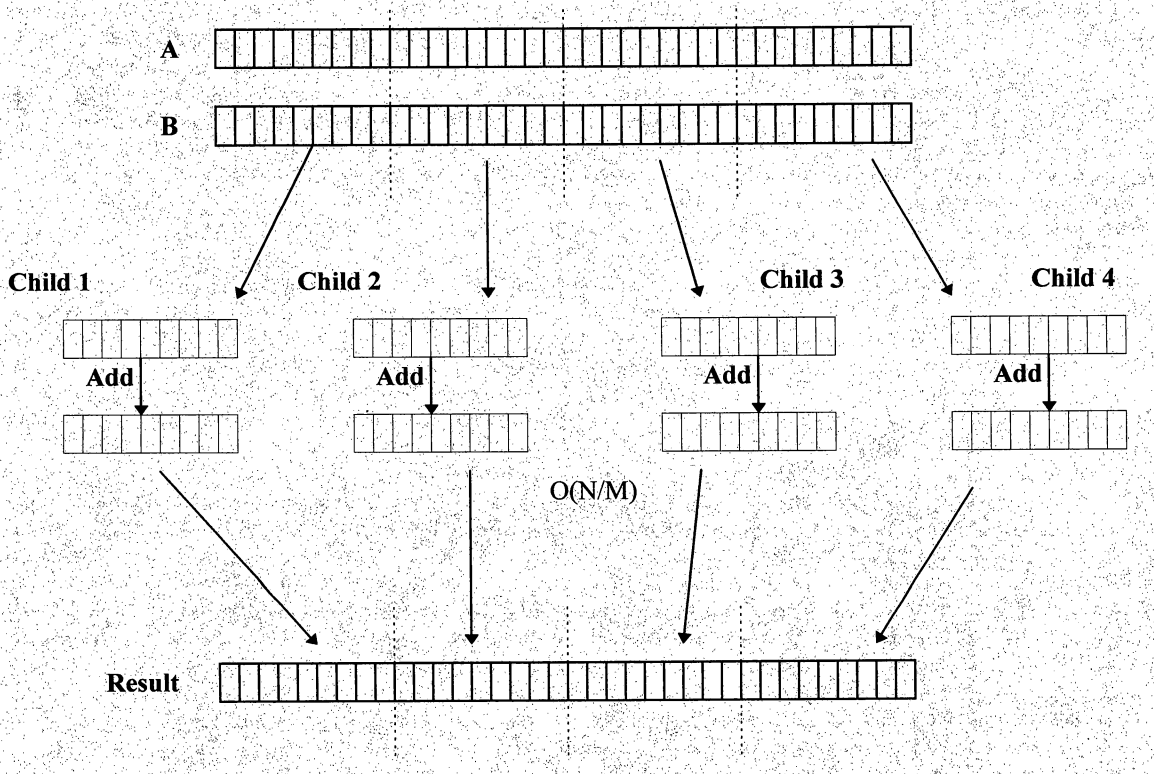

Figure 4.11 Distributed Vector Addition.

The transmission time is a major issue during computation, and we will discuss this in Section 4.3.

4.3 Performance and Analysis

The main purpose of the distributed computation in the Spider System is to provide a facility for submitting computing jobs without worrying about resources on how the job is going to be executed. The entire implementation is done in user level; therefore, there are unavoidable overheads in communication and data transmission.

The performance of the distributed computation will be analyzed and compared with Parallel Virtual Machine (PVM) [7], the SGI Power Challenge, and a single SGI workstation.

4.3.1 Comparison with PVM

The major costs in the implementation of Distributed Computation Service are the time of data transmission in the network, setting up remote server objects (OSB and server object activate the remote objects), and talking to TM. We compare the control messages needed to be exchanged in the Distributed Computing Service with PVM system. The control messages are for tasks and objects (daemons in PVM) to set up all required information before sending data. Figure 4.12

shows the comparison of the control messages exchanged in these two systems.

Distributed Computation Service in Spider

Where	send and recv
OSB-Registry	3
OSB-Task Manager	6
SVR_OSB-Task Manager	11N
Total	9+11N

*** N is the number of server objects that request for children from TM.**

(a)

Parallel Virtual Machine

Where	send and recv
Task-Task	0
Task-pvmd	2
pvmd-pvmd	2N
Total	2N + 2

*** N is the total number machines running pvmd.**

(b)

Figure 4.12 Comparison of the control messages.

According to the tables in the Figure 4.12(a), every time a user submits a job to the Distributed Computation Service, there are 9 control messages to be exchanged before starting the execution. PVM needs to exchange 2N control messages when there are N machines are added to the PVM system, and each spawned task needs to contact its local

pvmd via the libpvm library in order to register the task into PVM (see Figure 4.12(b)). After PVM finishes the initialization, the master pvmd will send a message to each slave pvmd periodically to check its status.

The major difference between PVM and the Distributed Computation Service in the Spider System is that PVM is not in object-orientation, but the DCS is. Thus, DCS is more flexible and intelligent that DCS is able to locate the available servers dynamically and choose the most appropriate machines for the users application. Whereas in PVM, the user must specify which machines will be included in the PVM system, and PVM activates slave pvmd daemons on all the machines specified by the user.

4.3.2 Performance Results

In this implementation, we test and collect the execution time for different size of data. All tested data are generated from random numbers. The following comparisons are divided into two categories: real time and user time. The real time is the wall clock time. The user time is the CPU time of a program execution in the local machine.

One of the SGI workstations and the SGI Power Challenge machine are chosen to test the performance on a single machine. DCS and PVM running on SGI workstations are tested.

4.3.2.1 Matrix Multiplication

The maximum size of matrix tested is 300 (300x300 of a matrix).

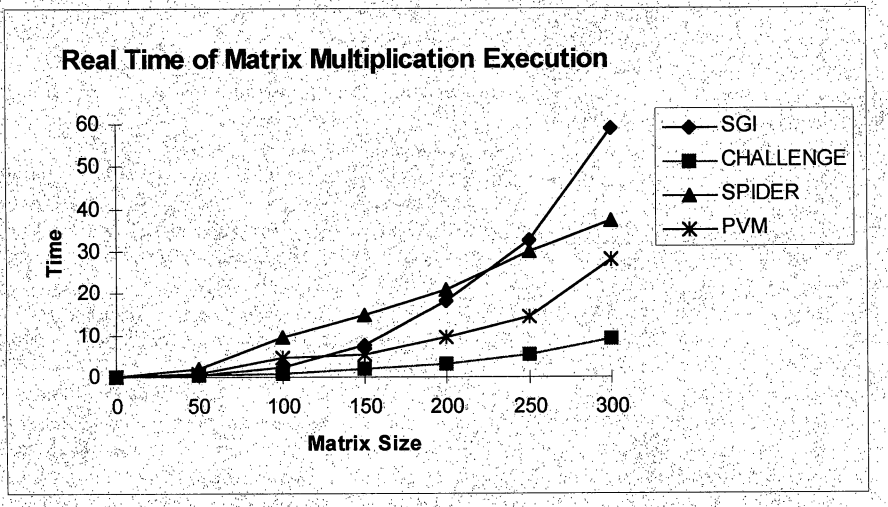

Figure 4.13 Comparison of Matrix Multiplication in real time.

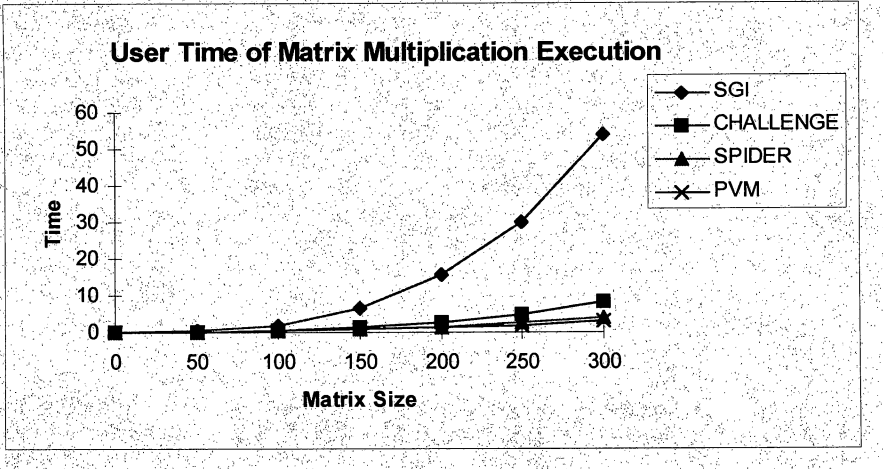

Figure 4.14 Comparison of Matrix Multiplication in user time.

In Figures 4.13 and 4.14, the average total execution time (real time) for a program submitted to the Spider DCS

is higher than the program executed on a single machine (SGI workstation and Power Challenge), but the average user time for both DCS and PVM are much smaller than a single SGI workstation.

4.3.2.2 Vector Addition

The maximum array size tested was 100,000 in the vector addition. Figures 4.15 and 4.16 show the comparisons of the average real time and user time for different systems.

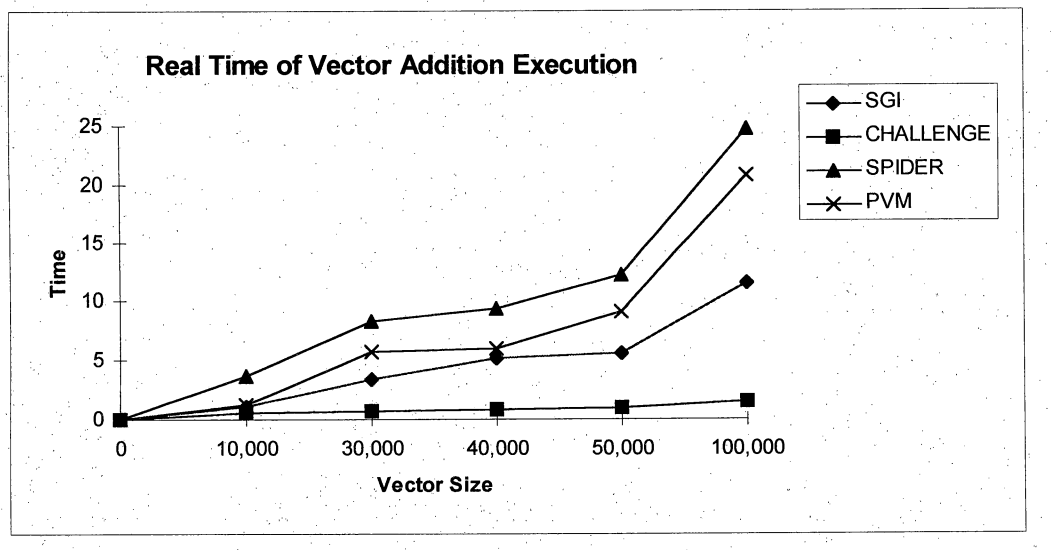

Figure 4.15 Comparison of Vector Addition in real time.

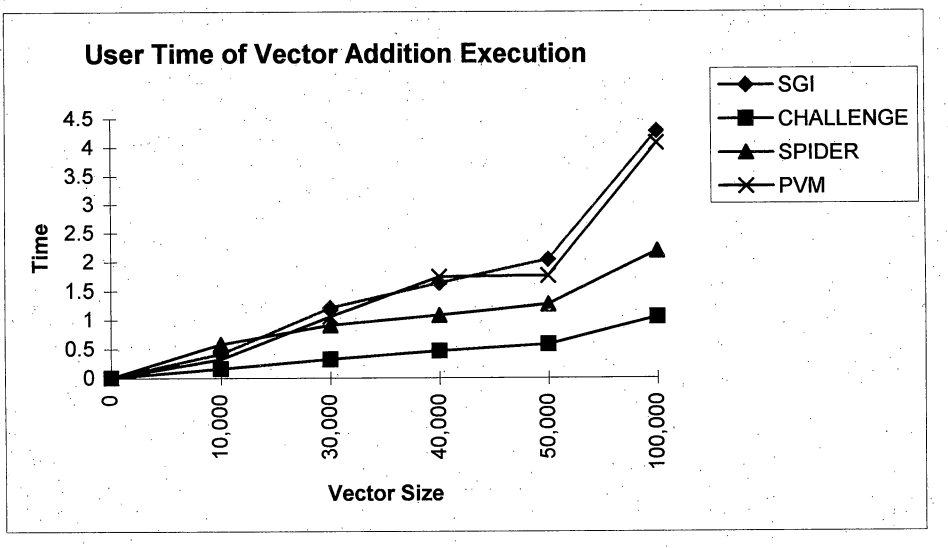

Figure 4.16 Comparison of Vector Addition in user time.

In this test, the performance of programs executed on a single machine has a shorter average real time than DCS and PVM. For the average user time, DCS has better performance than PVM and a single SGI workstation.

4.3.2.3 Distributed Quicksort

The maximum data size tested was 100,000 in the distributed quicksort.

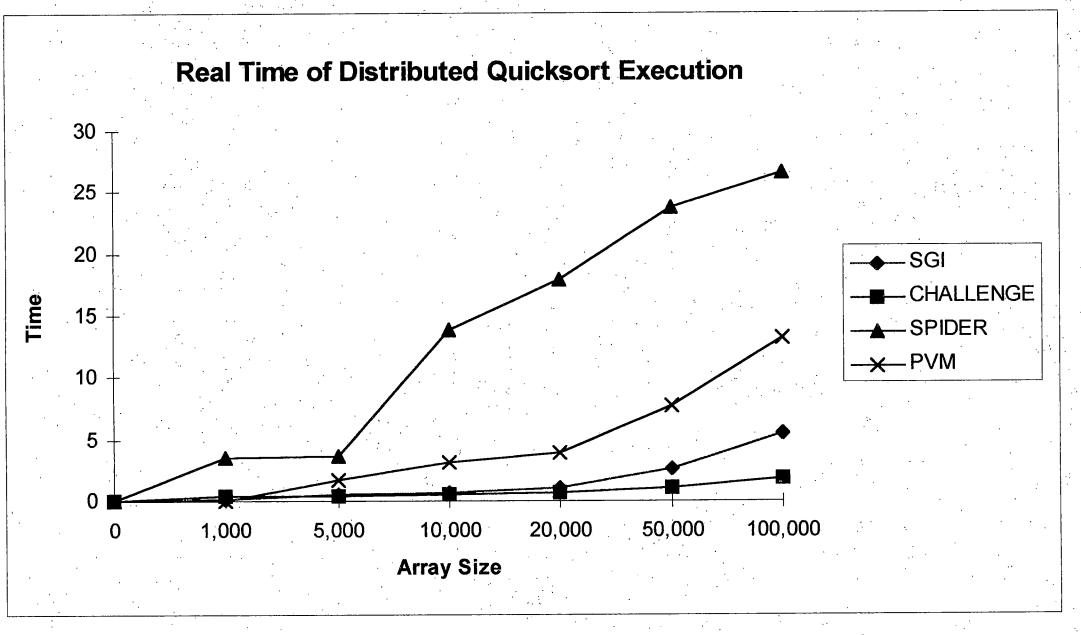

Figure 4.17 Comparison of Distributed Quicksort in real time.

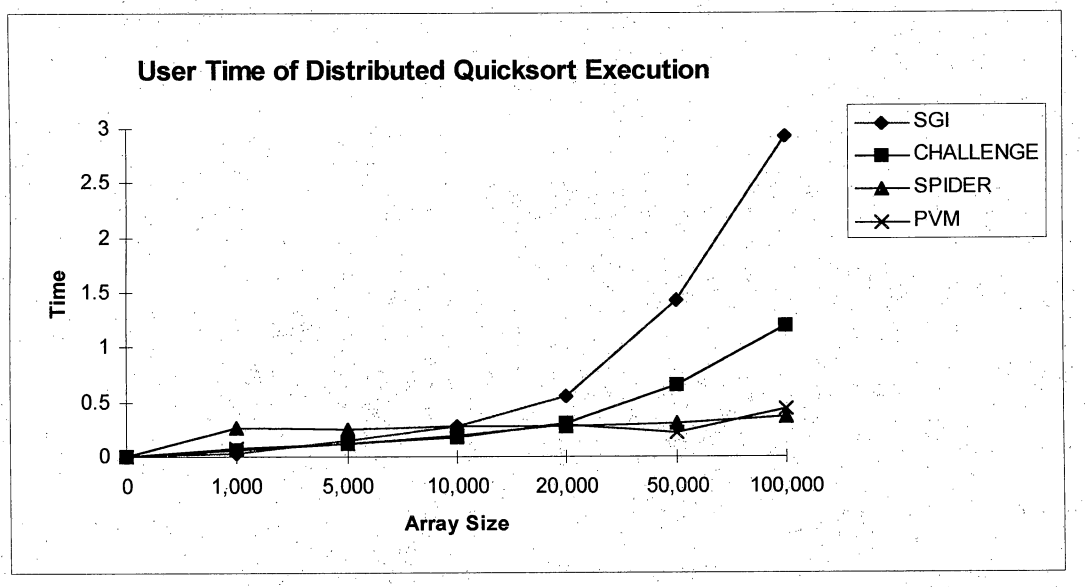

Figure 4.18 Comparison of Distributed Quicksort in user time.

In Figure 4.17, DCS failed to provide a good performance for real time test, where DCS had the highest real time than the other three systems. On the other hand, DCS and PVM have a very competitive performance in user time, see Figure 4.18.

4.3.3 Analysis

Figure 4.19 shows who has the best performance in the group. The Silicon Graphics Power Challenge XL has surpassed Spider, PVM, and a SGI workstation in most performance tests. When those test programs were executed on the Power Challenge machine, each program was partitioned and running on four CPUs in a true parallelism; therefore, Power Challenge showed the best performance in the overall test. PVM has an impressive result in the matrix multiplication. Although the Spider's DCS fails to improve the program execution time, DCS still improves the user time in all three tests.

Test / Machine	Matrix Multiplication		Vector Addition		Distributed Quicksort	
	Real Time	User Time	Real Time	User Time	Real Time	User Time
DCS						
PVM		X				X
CHALLENGE	X		X	X	X	
SGI Workstation						

X: indicates the system that has the best average time among the group.

Figure 4.19 Performance analysis table.

The major costs of the performance in the DCS are data transmission on network, remote object activation, and waiting for TM's services. Current network configuration in the CSnet (see Figure 2.1) relies on the Ethernet, which can only provide 10 Mbps bandwidth. To activate the remote server object, the *rsh* function call is used which contain some overhead when invoking a Unix system call. Furthermore, the time spent on waiting and talking to Task Manager, especially in distributed quicksort computation where a number of children accessing the TM simultaneously, is the main factor of the slow performance.

To predict the estimated overhead in the Distributed Computation Service, we analyze the time spent in each

computation service and generate equations to express the overhead of our implementation. The following sections explain the estimation of overhead in DCS.

4.3.3.1 Analysis of Distributed Quicksort

In the distributed quicksort implementation of DCS, we estimated the overhead in its execution. The following explains the meaning of each symbol used in this analysis:

T_o: the average time to start a remote object.
T_d: the average time for a node to pack and unpack data.
T_p: the average time for transmitting one packet of message.
T_w: the average waiting time to get the service from TM
N: the data size of integers.
M: the maximum number of elements* that can be performed using quicksort in one node.
P: the number of elements of each packet (maximal elements to be transmitted at a time).
C: the maximum number of leaf nodes.
L: the level of the link.
L_T : total levels of the links in the tree.
*Note: each element of the data is an integer and has 4 bytes representation in Unix systems.

$$L_T = \lceil \log_2(N/M) + 1 \rceil \qquad C = 2^{L_T-1}$$

Each link between two nodes has the following approximate number of data transmission:

$$2*(N/(2^{L-1}*P)) + 5$$

$(N/2^{L-1})$ is the size of data given to the child node and 5 is the number of messages exchanged for other information. The total transmission time for a link from start to finish is

$(2*(N/(2^{L-1}*P)) + 5) * T_p$

Each parent spends T_w when it needs children by talking to TM. Thus, the estimated transmitting time of all links can be expressed as follow:

$$\sum_{i=2}^{i=L_T}[2^{i-1}*((N/(2^{i-2}*P)+5)*T_p) + T_o + T_d] + \sum_{i=0}^{i=L_T-2}2^i * T_w \qquad (1)$$

The control messages for the execution is 9+11n, where the n is the number of parent nodes. The transmission time for the control messages is:

$$[9 + 11 * (2^{L_T-1}-1)] * T_p \qquad (2)$$

According to the equation (1) and (2), we can find the estimated time of overhead for a program in the DCS as:

$$T_{total} = [9 + 11 * (2^{L_T-1}-1)] * T_p + T_o + \sum_{i=2}^{i=L_T}[2^{i-1}*((N/(2^{i-2}*P)+5)*T_p) + T_o + T_d] + \sum_{i=0}^{i=L_T-2}2^i * T_w \qquad (3)$$

where $(9+11*(2^{L_T}-1))$ is the number of control messages exchanged, and the T_o is for the first server object. For example, giving the following measured values:

T_d is 0.02 second
T_p is 0.018 second
N = 20000
P = 250
T_o is 0.84 second
T_w is 0.79 seconds
M = 5000

then, $\mathbf{L_T} = \lceil \log_2(N/M) + 1 \rceil = 3$ and $\mathbf{C} = 4$.

Thus,

$$\mathbf{T_{total}} = \mathbf{(9+11*3)*T_p + T_o +}$$

$$\sum_{i=2}^{i=L_T} [2^{i-1} * ((N/(2^{i-2} * P)) + 5) * T_p) + T_o + T_d] + \sum_{i=0}^{i=L_T-2} 2^i * T_w$$

$$= \mathbf{11.99} \text{ seconds.}$$

This $\mathbf{T_{total}}$ is the total estimated overhead during execution. If we add this to the time that a program spends in initialization, computation, and printing message to the terminal, the total execution time will be about 13-15 seconds. The average time for initialization, computation, and printing messages during execution is 2.48 seconds. Thus, the estimated time 14.47 seconds. The actual execution time of the same data size in the test is 17.81 seconds. The estimated execution time is smaller than the actual time because we assume the network load is steady (the data transmission is a constant) and data is always partitioned into two same size of data. Therefore, the estimated time is approaching the best case of the computing time of Distributed Quicksort and is smaller than the actual measured time from the test. From this analysis, we are able to know that the main cost for distributed quicksort is data transmission, which takes over 80% of total execution time.

When the data size increases, we can expect that the estimated overhead will increase exponentially. In addition, we also need to consider the latency of the network load and the object's machine load during the execution. These factors also can affect the slow performance of DCS.

4.3.3.2 Analysis of Distributed Matrix Multiplication

Since the basic computing model of distributed matrix multiplication and vector addition are similar to each other, we present the analysis of estimated overhead of distributed matrix multiplication in this section.

The meanings of symbols in the analysis are explained as follow:

T_o: the average time to start a remote object.
T_d: the average time for a node to pack and unpack data.
T_p: the average time for transmitting one packet of message.
T_w: the average waiting time to get the service from TM
N: the size of a matrix (NxN)
N_{min}: the minimum size of a matrix that the multiplication is performed locally.
M: the size of a sub-matrix (MxN)
P: the number of elements of a packet (maximal integers to be transmitted at a time).
C: the maximum number of children.
*Note: each element in a matrix and a packet is an integer and has 4 bytes representation in Unix systems.

When we calculate the multiplication of two matrices, for example, $A_{NxN} \bullet B_{NxN}$, A is going to be partitioned into sub-matrices, if N is greater than N_{min}. If $N > N_{min}$, then

there will be **C** = $\lceil N/M \rceil$ children to be spawned. Each link between parent and child has the following number of data transmission:

$$(2*M*N + N^2)/P + 5$$

Because each child only calculate the multiplication of $A'_{MxN} \bullet B_{NxN}$, the size of the product is MxN.

Thus, the total estimated transmission time for the distributed matrix multiplication can be expressed as follow:

$$C * [((2*M*N + N^2)/P + 5)* \mathbf{T_p} + \mathbf{T_o} + \mathbf{T_d}]$$

The control messages for the execution is 20, because in the computing model of matrix multiplication and vector addition there is only one parent during execution. The transmission time for the control message is 20 * $\mathbf{T_p}$.

Therefore, the estimated overhead for a matrix multiplication program executed in the DCS is:

$$\mathbf{T_{total}} = 20*\mathbf{T_p} + \mathbf{T_o} + \mathbf{T_w} + C*[((2*M*N + N^2)/P + 5)*\mathbf{T_p} + \mathbf{T_o} + \mathbf{T_d}]$$

For example, giving the following measured values:

$\mathbf{T_d}$ is 0.02 second
$\mathbf{T_p}$ is 0.016 second
N = 200
P = 250
$\mathbf{T_o}$ is 0.84 second
$\mathbf{T_w}$ is 0.3 second
M = 40
$\mathbf{N_{min}}$ = 50

then, C = 5.
Thus,

$$\mathbf{T_{total}} = 20*\mathbf{T_p} + \mathbf{T_o} + \mathbf{T_w} + C*[((2*M*N + N^2)/P + 5)*\mathbf{T_p} + \mathbf{T_o} + \mathbf{T_d}]$$

$$= 24.08 \text{ seconds}$$

This T_{total} is the total estimated overhead during execution. If we add this to the time that a program spends in initialization, computation, and I/O , which is 1.04 seconds as an average, the total estimated execution time is 25.12 seconds. This estimated result is close to the actual average execution time of 21.07 seconds.

When the matrix size increases, we can expect that the estimated overhead will increase linearly. The overhead for distributed vector addition also has the same estimated overhead tendency as the distributed matrix multiplication.

4.4 Conclusions of the Implementation

The Distributed Computation Service for the Spider Project is designed and implemented. This implementation shows that the DCS provides a transparent computation for user applications. The performance of the DCS may not show good results, but the performance is not the main concern of this implementation. The DCS is able to reduce the CPU usage of the user's local machine by distributing jobs to other machines. Furthermore, this computation model can be used to solve other complex problems in parallel and distributed manner, such as NP-complete problems (e.g., the Traveling Salesman Problem and finding Hamiltonian cycle).

PVM is a daemon-based software application, every machines in a parallel virtual machine are inter-connected by its local pvmd that speeds up the tasks' execution. Also, all data transformation and representation are built in the libpvm library, which offers the ease for users in programming. The prototype of the DCS in Spider can be optimized and improved by the following suggestions:

a) optimize the amount of messages exchanged during execution by using packet-oriented methodology;
b) provide an application program interface for the ease of porting users programs to use DCS; and
c) reduce the costs of data transmission by using the FORE ATM backbone network which can transmit data at 155 Mbps.

CHAPTER 5 Conclusions and Future Directions

We have shown the specification and design of the Spider System and the defined functionalities for the Spider in an object-oriented approach (OOA). We identified five functionalities for the Spider System and they are: distributed file system, security, clock synchronization, scheduling service, and distributed computation service. The implementation of the Distributed Computation Service (DCS) has shown a reasonable performance from our performance results. This chapter will conclude the work for this thesis and give future directions.

5.1 Conclusions for the Specification and Design of the Spider Project

In this thesis, the basic structure and main components (Microkernel, OSB, and Functionality Server) of the Spider System are designed and given the specification of five functionalities. All the components and functionalities in the Spider must be in object-orientation, in order to integrate a truly object-oriented distributed system. Using OOA, Spider facilitates easy maintenance, modification, and simplicity in adding more features in the future.

From the survey of Sun's Spring [14], OSF's DCE [17, 23], PVM [7], and Condor [24], we are able to adopt their strengths for our design. The Spider Project and the implementation of DCS will provide a testbed for people to do related research on parallel and distributed computation. The derivations of those functionalities defined for Spider are described as follow:

a) The Distributed File System (DFS) of Spider is designed to provide a distributed file storage which is different than monolithic systems (e.g., UNIX). The Naming Server, Directory Server, and Caching Server of DFS are defined to provide a transparent and coherent file accessing for users. The strengths of Spring File System [15] and DCE's distributed file service [17, 23] are adopted for designing the DFS of Spider.

b) Security in Spider is designed to protect the whole Spider System from any unauthorized and inappropriate access of objects. The Authentication Server and Access Control List Server are defined for users to access any object that they are permitted to use. Also, we adopted the strengths from Spring's security [14] and DCE's security

service [17, 23] for the design of the security in spider.

c) Scheduling service in Spider is designed to offer intelligent services for user applications. The Scheduling Manager and Pooling Server are designed to migrate user's application on one or more idle and/or low-utilization workstations. Users are able to utilize as many workstations as their applications needed for computation from available machines. The concept from the Condor scheduling system and the scheduling algorithm [13, 14] are adopted to design the Scheduling service of the Spider System.

d) Distributed Computation Service adopts the basic design of PVM [7] to provide a transparent and distributed computation for user applications.

5.2 Conclusions for the Distributed Computation Service

By using the specification and design of the Spider System, we implemented the Distributed Computation Service for Spider. In the current implementation of DCS, the three services are provided -- distributed quicksort, distributed matrix multiplication, and distributed vector addition. These services employed the conceptual object-oriented

design of the Spider Project, which includes Object Service Broker, Task Manager, and Registry Server to assist the Distributed Computation Service.

PVM [7] shows an excellent work for programs executing in a heterogeneous computing environment which is able to collect from single-CPU machine to multiprocessors as part of virtual machine. A set of programming interfaces and libraries are provided for user applications. To provide fault tolerance, PVM allows user program to delete or add machines during operation. However, users need to configure each machine's environment variables for PVM daemon and libraries, and users may need to add all machines to be included by the PVM or submit a host file to PVM when starting up PVM. After the virtual machine is configured, users may run their PVM programs in a distributed and parallel manner.

The major difference of DCS and PVM is that DCS does not require users to specify that which machines are going to be included in the DCS. The local Registry Server and Task Manager can handle all available resources for user applications. OSB is able to locate the remote machines and activates remote server object to perform computation. Users have no knowledge of knowing which machines will be used for the program computation. To avoid increasing a machine's

workload, DCS does not assign any workstation to be used more than once. This can ensure to get a faster computation by limiting each machine's workload. In PVM, it is possible for a machine to be spawned more than once during computation.

DCS also provides the fault tolerance. A mirror Task Manager server backups all task activities from the Task Manager server in order to continue monitoring program execution, if the Task Manager server fails.

Figure 4.19 shows the performance comparisons of DCS. Since the major costs of slow performance in DCS are data communication and network transmission, DCS may not show a faster computing performance. However, it reached the objective of reducing the user time in computation and provided a testbed for users to do research on distributed and parallel computation. Furthermore, the DCS is able to provide a transparency for user applications and fault tolerance for computations.

The technology of networking is improving every year. If the network used in the DCS implementation is replaced by ATM (OC-3c) or OC-12 [23], the overhead of data communication and transmission will be reduced to a minimum of time and the performance for the DCS will be greatly improved.

5.3 Future Directions

The Spider Project is at its initial state, only the Distributed Computation Service is implemented. We hope that other researchers continue the work and implement the rest of the defined functionalities. There are several directions needed to be pursued in the future:

a) The microkernel is the heart of the Spider Project; however, it is also the most complex component of the Spider. The major concern of constructing the microkernel is how to include the various system architectures in the CSnet.

b) To support a distributed object system, the IDL for Spider should be defined and implemented in order to provide a strong interface for all objects. We can adopt the specifications of DCE's IDL [17] or OMG's IDL [16] and implement Spider's IDL in the future.

c) Each defined functionality in the Spider Project gives a direction for future research. There maybe new methodology and technology in the future. How to adopt the new methodology and technology for the Spider system and determine the needs of our department become the most important directions in the future.

d) Improve the Distributed Computation Service by implementing a programming interface whereby a given program can be partitioned into components so that they can be executed transparently in a parallel and distributed manner. Furthermore, a graphical application tool can be implemented in order to provide easy manipulation and transformation tools for original programs into the equivalent distributed computing programs.

REFERENCES:

[1] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. *Mach: A Kernel Foundation for Unix Development*. Proc. Summer 1986 USENIX Conference, USENIX. pp. 93-112, 1986.

[2] T. E. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. *Severless Network File Systems*. ACM Trans. On Computer Systems, Vol. 14, No. 1, February 1996, pp. 41-79.

[3] A. Birrel and B. J. Nelson. *Implementing Remote Procedure Calls*. ACM Transactions on Computer Systems, vol. 2, no. 1, Feb. 1984, pp. 39-59.

[4] F. Cristian. *Probabilistic Clock Synchronization*. Distributed Computing, vol.3, pp. 146-158, 1989.

[5] R. DuVall. *DuVall's Simulated Global Clock: A Messaging Approach for a Distributed System.* Research paper of CS624 at CSUSB, Fall, 1996.

[6] M. Gien. *Micro-Kernel Design*. UNIX REVIEW, 8(11): pp. 58-63, November 1990.

[7] A. Geist, A. Beguelim, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine*. The MIT Press, Cambridge, 1994.

[8] D. Golub, R. Dean, A. Forin, and R. Rashid. *Unix as an Application Program*. Proc. 1986 Summer USENIX Conference, pp. 87-95, Anaheim, California, 1986.

[9] G. Hamilton and P. Kougiouris. *The Spring Nucleus: A Microkernel for Objects*. Proc. 1993 Summer USENIX Conference, pp. 147-160, June 1993.

[10] G. Hamiliton, M. L. Powell, and J. G. Mitchell. *Subcontract: A Flexible Base for Distributed Programming*. Proc. 14th ACM Symposium on Operating Systems Principles. Pp. 69-79, December 1993.

[11] J. Howard, M. Kazar, S. Menees, D. Nichols, S. Satyanarayanan, R. Sidebotham, and M. West. *Scale and Performance in a Distributed File System*. ACM Trans. On Computer Systems, 6(1):51-81, Feburary 1988.

[12] L. Lann. *Distributed System—Towards a Formal Approach*, Information Processing Letter, North-Holland, vol. 77, 1977, pp. 155-160.

[13] M. J. Litzkow, M. Livny, and M. W. Mutka. *Condor - A Hunter of Idle Workstations*. Proc. the 8th International Conference on Distributed Computing Systems, June 1988, pp. 104-111.

[14] J. G. Mitchell, J. J. Gibbons, G. Hamilton, P. B. Kessler, Y. A. Khalidi, P. Kougiouris, P. W. Madany, M. N. Nelson, M. L. Powell, and S. R. Radia. *An Overview of the Spring System*, Proceedings of Compcon Spring 1994, February 1994.

[15] M. Nelson, Y. Khalidi, and P. Mandany. *The Spring File System*. Sun Microsystems Laboratories, Technical Report SMLI TR-93-10, February 1993.

[16] OBJECT MANAGEMENT GROUP. *The Common Object Request Broker: Architecture and Specification, Revision 2.0*. Tech. Rep. OMG TC Document 93-03-04, The Object Management Group, Framingham, MA, 1995.

[17] OSF. *Introduction to OSF DCE*. Englewood Cliffs, NJ: Prentice Hall, 1992.

[18] M. J. Park. *An Optimistic Concurrency Control Based on Clock Synchronization*. Master's Thesis, CSUSB, 1996.

[19] J. Rumbaugh, M. Balha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1991.

[20] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. *Design and Implementation of the Sun Network Filesystem*, Proc. Of the Summer USENIX Conference, 1985, pp. 119-130.

[21] M. Singhal and N. G. Shivaratri. *Advanced Concepts in Operating Systems*. Mc-Graw-Hill, Inc. New York, 1994.

[22] J. Steiner, B. Neuman, and J. Schiller. *Kerberos: An Authentication Service for Open Network Systems*. Proc. 1988 USENIX Conference, Winter 1988.

[23] A. S. Tanenbaum. *Distributed Operating Systems*. Prentice-Hall, Inc., Upper Saddle River, New Jersey, 1995.

[24] T. Tannenbaum and M. Litzkow. *The Condor Distributed Processing System*. Dr. Dobb's Journal, February 1995, pp. 40-48.