A DISTRIBUTED SYSTEM FOR FUNCTIONAL COMPUTING

By

ZHILAN LIU

Bachelor of Engineering

Northwest Fangzhi University

Xi'an, China

1992

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May, 2001

A DISTRIBUTED SYSTEM FOR FUNCTINAL

COMPUTING

Thesis Approved:

[illegible]

Thesis Adviser

[illegible]

[illegible]

[illegible]

Dean of the Graduate College

ACKNOWLEDGEMENTS

I would like to express my sincere appreciation to my advisor, Dr. K. M. George, for his continuous guidance, concern, patience, and support throughout my graduate program. I shall always be thankful for his inspiration and advice on my study of computer science, his encouragement and help for me to apply computer science theory to practice. I would also like to express my gratitude to my thesis committee members, Dr. G. E. Hedrick, and Dr. N. Park for providing valuable suggestions and ideas.

I wish to thank all the faculties and staff members of Computer Science Department who are always kind and helpful to me. I extend my thanks to Oklahoma State University for giving me the opportunity and providing me financial support to accomplish my graduate program.

I am grateful for the help I have gotten from many individuals. In particular, I wish to extend my gratitude to Dr. Ronald Beer, Dr. Hon-shiang Lau, Dr. Amy Lau, Mr. David Bryan, Dr. Huizhu Lu, Dr. Fang Zhu, and Ms. Kathy Burnap for their great help in my life. I also express my deepest gratitude to my friends, Yali Zhu, Jianhua Ren, Xiaoduan Sheng, Sha Li, and Xinhua Wang always gave me generous help and encouragement during my stay in OSU.

Finally, but not least, my appreciation goes to my parents, Mr. Tianding Liu and Mrs. Rongjing Lei, and my elder sister, Hong, for their deep love and full support. This thesis is dedicated to them without reservation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

CHAPTER 1

INTRODUCTION

The demand for high throughput with relatively low cost and the ability to handle very large amounts of data have been forcing computer system designers to consider nontraditional architectures such as parallel/distributed systems. In an effort to overcome some of the limitations of a uniprocessor system one of the major development in computing in recent years has been the introduction of a variety of parallel computers. Parallel, distributed, concurrent or multiple computes are such that there is more than one processing unit within the system, thereby allowing the device to execute many tasks simultaneously [1].

This requirement of parallel/distributed computation has prompted the researchers to find ways that can accommodate parallelism differing from Von Neumann computers. Fortunately, it is possible to integrate data communications and sequential Von Neumann computers to form either a wide area network (WAN) [1] or a local area network (LAN) [1]. This may be regarded as a network form of a multiple processor system. Such a system uses message passing as its data mechanism. These systems are usually referred to as either computer networks or distributed computing systems.

Generally, distributed computation systems can be classified as distributed computation programming models and distributed computation applications. Most of the programming models and applications use TCP/IP [2, 3, 4] and the OSI model [2, 3, 4] as their basic protocols in a distributed networked environment. RPC (Remote Procedure Call) [5], DCE (Distributed Computing Environment) [6], CORBA (Common Object

Request Broker Adapter) [7] and Java RMI (Remote Method Calls) [8] are currently available programming models for the different network components to work in a hardware independent environment. Based on these programming models client-server model [9] and mobile agents [10, 11] have been implemented as distributed computing applications [12]. Several application systems have exploited the advantages of distributed computation. Condor [13, 14] and Piranha [15, 16] implement environments for adaptive execution of parallel computations. To achieve adaptive parallelism over a set of dynamically changing processors, these systems utilize idle-state workstations. DDAS [17, 29] environment is somewhat related to Condor and Piranha [17], but provides better security, because its domains are only those computers, which can be accessed by a user with full authorization. It also supports adaptive network because nodes can be added or removed as necessary. However, there are some limitations of the current design and implementation that makes it unsuitable for further use.

First, all the remote machines have to have exactly the same user ID, password, and the path. The DDAS components are prematurely bound to these specific values. For example, when a host of the DDAS system updates and the path has been changed, the DDAS system will fail to construct a distributed system. Therefore, the design is not flexible to accommodate changes in remote system configurations.

Second, DDAS kernel components are mixed together with DDAS manager, which makes its functions unclear and dependent.

Third, when the DDAS system is constructing a network, if a host of the DDAS system node is down, the DDAS system freezes. That is, the system is not designed to be fault-tolerant.

Fourth, the DDAS routing has to be done manually, which forces the programmer to build several default routing files before constructing the system.

Last, the DDAS input command set can not guarantee the remote processes be killed, which may cause deadlock.

The first two cases are caused by flaws in the design of the DDAS kernel components and the last three are caused by the design of its manager. Therefore, a modified DDAS system is needed to be developed.

In this thesis, we modify DDAS to remove the above mentioned problems. We also design an environment for functional style distributed computing based on the distributed computing model and FP style functional programming [18,30].

The remainder of this thesis is divided into the following chapters:

Chapter 2: A detailed review of literature related to current distributed computation systems.

Chapter 3: Describes the FDCS model, including the FDCS model configuration, combining forms and illustrative examples.

Chapter 4: Discusses the design of the distributed system for composition and pipeline functional forms.

Chapter 5: Illustrates the implementation of the distributed system for composition and pipeline functional forms.

Chapter 6: Presents conclusion of the thesis and the future work.

Appendix A: Describes abbreviations and acronyms used in the thesis

Appendix B: Includes parts of system source code.

CHAPTER 2

LITERATURE REVIEW

Principally, there are two types of distributed systems: LANs and WANs. The main difference between the two is in the way they are distributed geographically. LANs are composed of processors that are distributed over small geographical areas, such as a single building or a few adjacent buildings. WANs are composed of autonomous processors that are distributed over a large geographical area (such as the United States). Furthermore, the WAN is used to make a network of LANs, called an internet. The OSI and TCP/IP models are used as the basic protocols in a distributed networked environment.

2.1 Higher-layer Network Protocols

Distributed programs rely on the operating system to provide facilities for synchronizing and communicating information among the constituent parts of the computation. Contemporary operating systems employ the OSI or TCP/IP reference architecture to implement communication at the upper layers of the network protocol stack.

2.1.1 OSI Model

The OSI (Open Systems Interconnection) model is based on a proposal developed by the International Standardization Organization (ISO) as a first step toward international standardization of the protocols. Even though the OSI model is used by very few organizations for network communication, it provides a stable hardware independent design. The OSI model has seven layers as shown in the right side of Figure 1.

2.1.2 TCP/IP Model

The TCP/IP (Transmission Control Protocol / Internet Protocol) model was originally developed by the U.S. DOD (U.S. Department of Defense). The TCP/IP protocols were designed to be independent of host hardware or operating system. The TCP/IP reference model has five layers as shown in the left side of Figure 1.

The Internet model can be thought of as a simplified OSI model. Figure 1 also depicts the relationship between the Internet model and the OSI reference model.

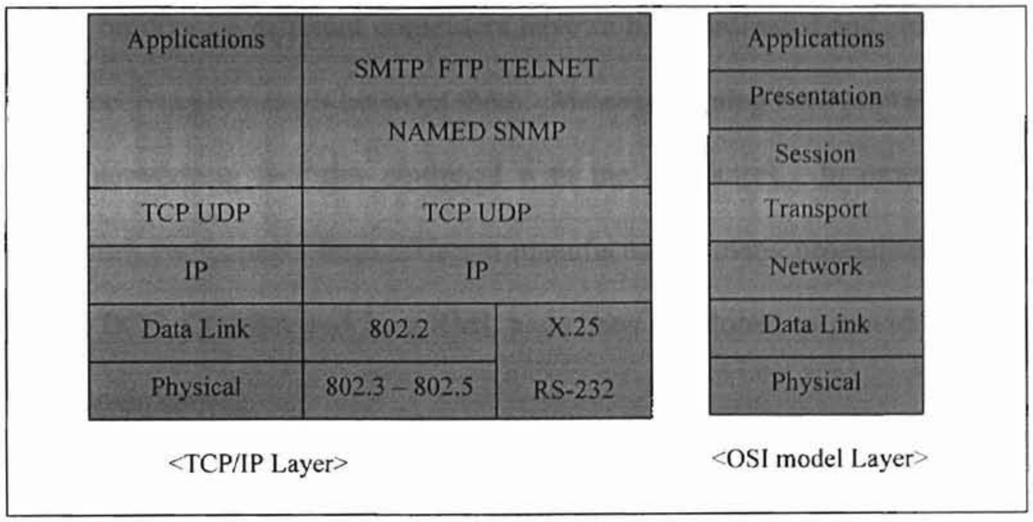


Figure 1. TCP/IP Layer and OSI Model Layer (Adopted from [2, 3, 4])

2.2 Distributed Computation Systems

Distributed computation refers to the case where a computation is implemented in such a manner that parts of the computation are executed on different machines (usually as processes); the parts of which use the network for communication and synchronization. In general, a distributed operating system is required to support distributed computations. Since the operating system must provide tools to accommodate various computing paradigms and to simplify the task of implementing a distributed computation, it generally uses the technology in its own implementation.

Generally, distributed computation systems can be classified as distributed computation programming models and distributed computation applications.

2.2.1 Distributed Computation Programming Models

Tasks running on different computers have to be coordinated and streamlined by sending and receiving messages between them. Message passing between computers can be done via networking facilities equipped with the computers. In order to support different network components from different manufacturers, many programming models, such as RPC, DCE, CORBA and Java RMI, have been developed to provide a hardware independent environment.

2.2.1.1 Remote Procedure Call

RPC was first investigated thoroughly by Nelson [19] in 1976 and has been in use in academic and commercial areas for many years [20]. By far, RPC is the most prevalent paradigm for structured client/server program development. Basically, it allows a program on one machine to call a subroutine on another machine without knowing that it is remote.

RPC is a specialized form of interprocess communication in which the initiating program performs a send operation immediately followed by a blocking read operation. The receiving program performs a blocking read until it receives the message sent by the "caller" process; it then provides the service and returns a result by sending it to the original process. From the original process's point of view, the IPC behaves as if it were a procedure call.

RPC implementations take the general form shown in Figure 2. The client machine consists of the client application code, a client stub, and the transport mechanism, while the server machine implements a transport mechanism, server stub, and the server code.

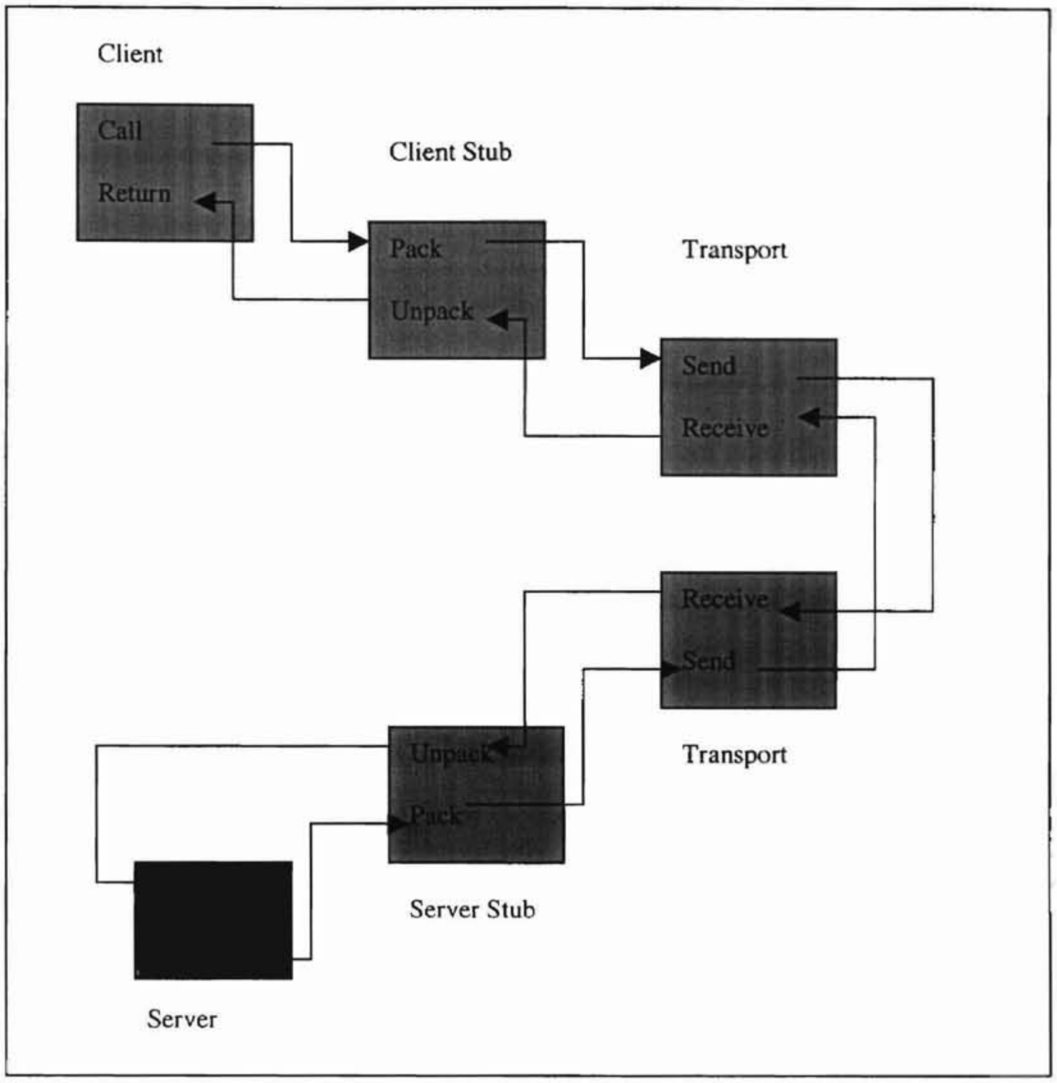


Figure 2. Remote Procedure Call

In the most simple form, the RPC is an obvious generalization of the traditional procedure call present in most programming language. The fundamental difference is that the calling procedure executes in one thread, process, or machine, and the called procedure executes in another. Because RPC is so analogous to the familiar mechanism, conceptually it is easy to construct applications using RPC. Most middle-ware such as DCE (Distributed Computing Environment), message queuing and network SQL (Structured Query Language) are built on RPC.

2.2.1.2 DCE

The OSF (Open Software Foundation) which was founded in 1988[21] describes DCE (Distributed Computing Environment). The DCE represents an attempt to gather base technologies from many vendors and make them part of an open environment. It is called middle-ware [22] or enabling technology. The DCE is a set of services and tools that support the creation, use, and maintenance of distributed applications in a heterogeneous computing environment. The remote procedure call (RPC) is an application programming paradigm and is part of OSF's DCE. It extends the local subroutine call semantics to a networked or distributed environment. Interfaces are defined in an Interface Definition Language, which is similar to C, and compiled and linked to the client and server parts of the application. The call by the client is packed then sent via the RPC protocol to the server, where it is unpacked and executed. Results are returned in a similar fashion. The RPC protocol includes calls for the client to locate and bind to the server, transmit data, and handle error conditions. RPC is designed to be independent of the transport protocol. Figure 3 describes the DCE architecture.

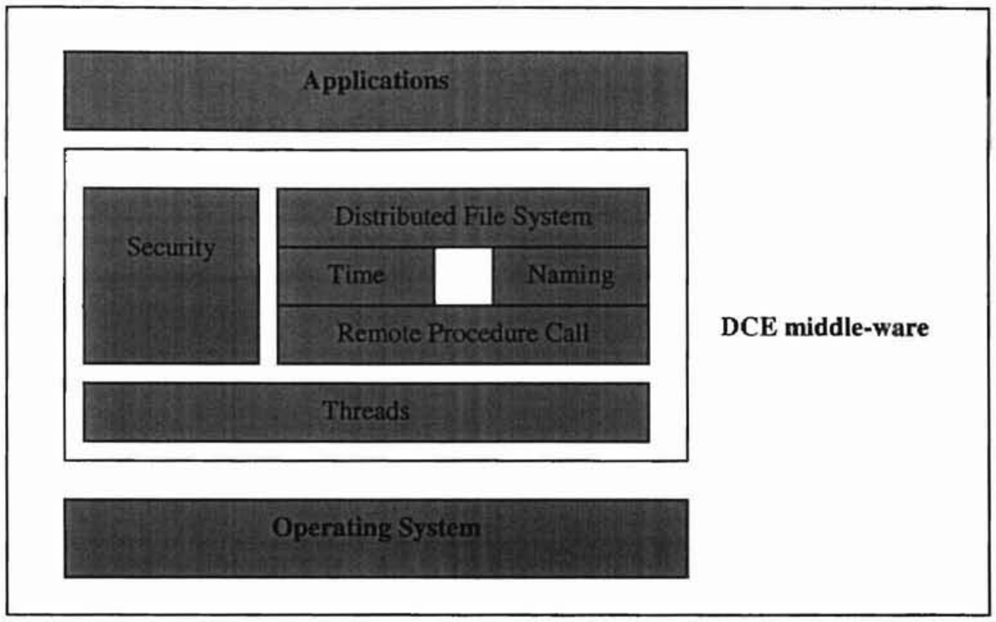


Figure 3. OSF DCE Architecture (Adopted from [23])

2.2.1.3 CORBA

CORBA (Common Object Request Broker Adapter) is another middle-ware platform in distributed computing environment. As a middle-ware, it is different from DCE in that CORBA uses an object-oriented distributed model whereas DCE utilizes a procedure-oriented distributed model [24].

The central component of CORBA is the Interface Definition Language OMG IDL. Programmers use OMG IDL to describe the interface to their objects. OMG IDL is the fundamental basis for the definition of the contract exposed by the object to the rest of the world.

Programmers who wish to develop CORBA objects must first begin by designing the OMG IDL for the objects they wish to create. Having completed the OMG IDL specification an implementor is then free to implement that language in any programming language, such as C, C++, Ada, Smalltalk and Java.

The actual CORBA architecture is depicted below in Figure 4.

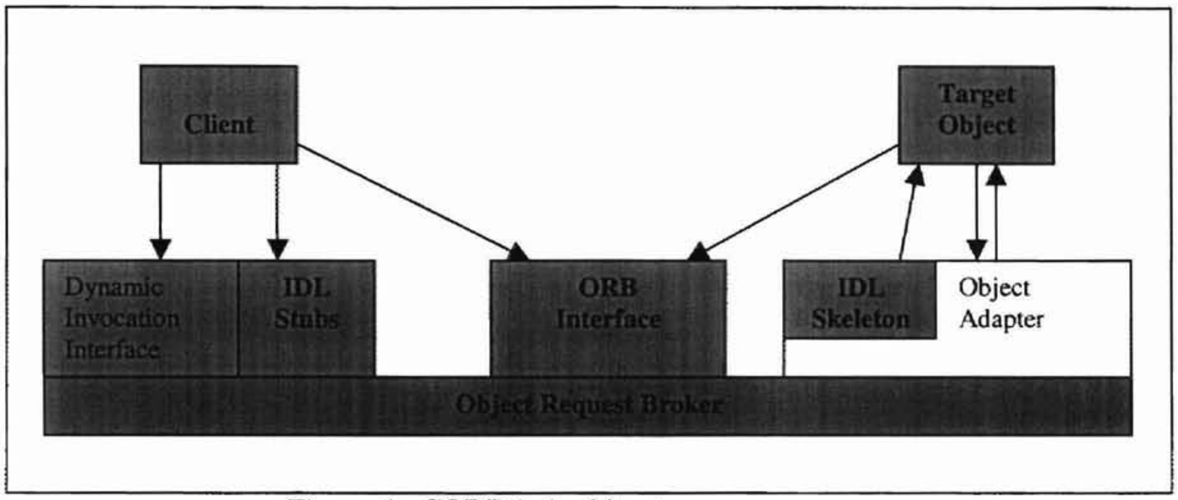


Figure 4. CORBA Architecture

In CORBA, objects are created in one location and remain at that location for a given lifetime. The entities which are passed over the network are "object references." An object reference is a unique identifier used to locate and describe a given instance of a given object type.

2.2.1.4 Java RMI

The Java Remote Method Invocation (RMI) package is a Java-centric scheme for distributed objects that is now a part of the core Java API. RMI offers some of the critical elements of a distributed object system for Java, plus some other features that are made possible by the fact that RMI is a Java-only system. RMI has object communication facilities that are analogous to CORBA's IIOP, and its object serialization system provides a way to transfer or request an object instance by value from one remote process to another. As shown in Figure 5, Java RMI consists of several layers and exists between applications and the JVM (Java Virtual Machine) [25].

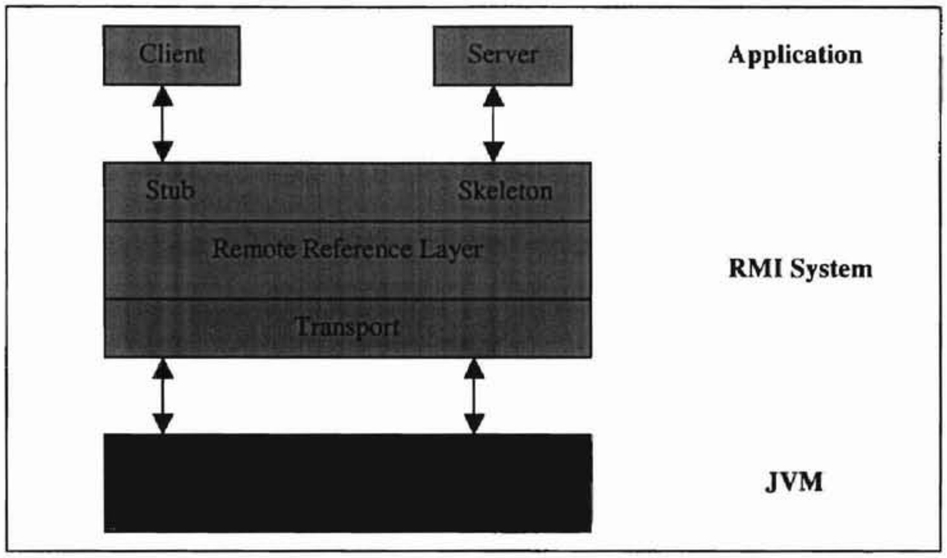


Figure 5. Java RMI Layers (Adopted from [26])

2.2.2 Distributed Computation Applications

CORBA and Java RMI serve as object-based distributed computation programming models for distributed computing applications, while RPC and DCE are used as the basis of client-server or mobile agents models in many applications.

2.2.2.1 Client-Server Model

The client-server model of computation is a popular distributed programming paradigm in which one process, the server, is a manager of one or more resources, a client, that is the user of the server's resources. A resource may be identified easily and physical in nature; e.g., a printer, or it may be hidden and abstract; e.g., an authentication database.

The client-server model is asymmetric, as suggested by the name. The persistent server "always exists in the network," passively waiting for requests for activity, while client processes decide when to utilize the server. A server is a slave process that solicits work, while a client is a master process that requires services. This can be seen in Fig 6.

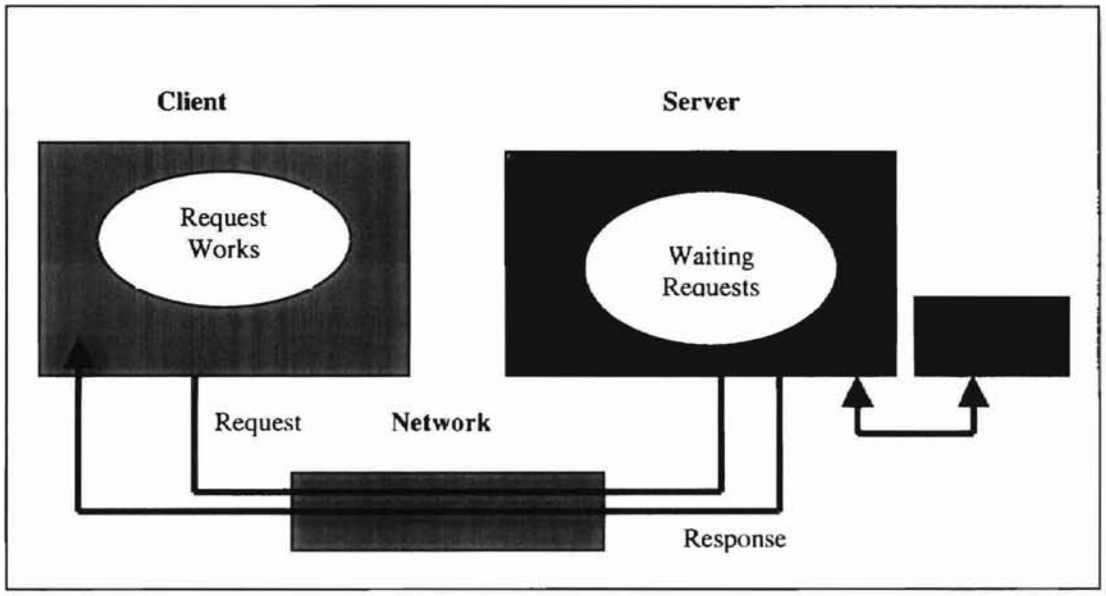


Figure 6. Client-Server Model (Adopted from [9])

2.2.2.2 Mobile Agents

Mobile agents are a convenient paradigm for distributed computing [27], since mobile agents are very efficient compared with the traditional approaches to distributed computation. The agent specifies when and where to migrate, and the system handles the transmission. This makes mobile agents easier to use than low-level facilities in which the programmer must handle communication explicitly, but more flexible and powerful than schemes such as process migration in which the system decides when to move a program based on a small set of fixed criteria.

"A mobile agent is a program that is able to change its location on its behalf and keeps its state (identity) across location changes" [28]. A mobile agent carries all of its internal state with it which eliminates the need for separate communication steps. The agent migrates to a machine performs a task, migrates to a new machine, performs a task that might be dependent on the outcome of the previous task and so on. In Figure 7, an agent carrying a mail message migrates first to a router and then to the recipient's mailbox. The agent can perform arbitrarily complex processing at each machine in order to ensure that the message reaches the intended recipient.

Mobile agents allow a distributed application to be written as a single program. Mobile agents can be viewed as extensions of the client/server model. Clients and servers can program each other and applications can dynamically distribute its server components when it starts execution.

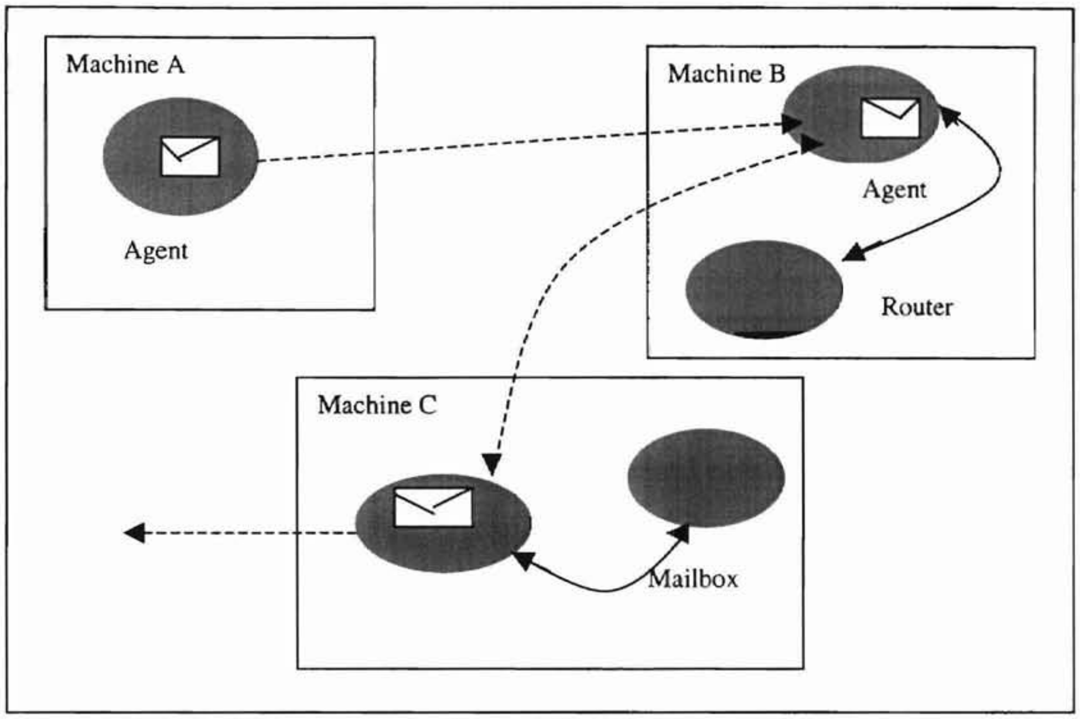


Figure 7. Operations of Mobile Agents System

CHAPTER 3

FDCS MODEL

This thesis is based on a functional distributed computing system (FDCS) that includes the advantages of distributed computing, contains reliable and secure transport system – DDAS [29], and takes advantage of the natural parallelism inherent in the FP programming model [30].

3.1 The FDCS Model Configuration

The FDCS model consists of three components (M, C, T) where M is a collection of modules, C is set of combining forms and T is a transport system. The combining forms are used to build new applications from existing ones. A module in M is a program f. In other words M consists of a collection of executable programs. f:x represents the application of f to x. The intuitive meaning is that f:x represents the result of running program f with input x. It is convenient to view the input and output as files (which actually contain the input and output respectively). With this assumption, we can assume that a program execution is the same as the application of a function to its argument file name and we assume that all functions are single argument functions. The set of combining forms constitute the heart of the system and provide the rules and constructs to build new software systems from the existing ones. The transport system supports distribution and network support. In FDCS, the DDAS is used to realize T.

3.2 Combining Forms

Combining forms are the heart of the system. They specify the semantics of the distributed system. An implementation is consistent with the semantics. One characteristic of this approach is that new combining forms may be added as required and thus making the system extensible. Here we describe several combining forms and their semantics. They are listed below

1) COMPOSITION: $f1 \bullet f2:x \equiv f1:(f2:x)$

f1 and f2 are programs that terminate and f1 executes when f2 terminates and the output of f2 is input of f1. This combining form imposes sequential control structure in the execution of programs. If f2 represents a collection of programs, this also serves as barrier synchronization.

2) PIPELINE: $f1 \parallel f2:x \equiv f1:(f2:x)$

f1 and f2 are programs that may execute concurrently. The input to f1 is produced by f2. This functional program captures pipeline parallelism.

3) CONSTRUCTION: [f1,f2, ..., fn]:x

f1...fn are independent (possible communicating) programs capable of executing in parallel. They could be distributed to different nodes of a network. There is no functional form designed for message passing communication. This form of communication can be accomplished by COMPOSITOIN or PIPELINE. Repetition is accomplished using the WHILE functional form. The result is assumed to be all list <x1, ..., xn> of outputs from the n programs.

4) APPLY_TO_ALL: f: <x1,...,xn>

Execute n copies of f; xi is the argument to the ith copy. This functional form captures data parallelism. The program can be distributed to different nodes in a network.

5) WHILE (c, f) : x

While condition c is true, repeat executing program f with input x. This functional form allows repetition.

6) IF (c, f, g) : x

If condition c is true, execute program f. Otherwise, execute program g. This functional form allows conditional execution.

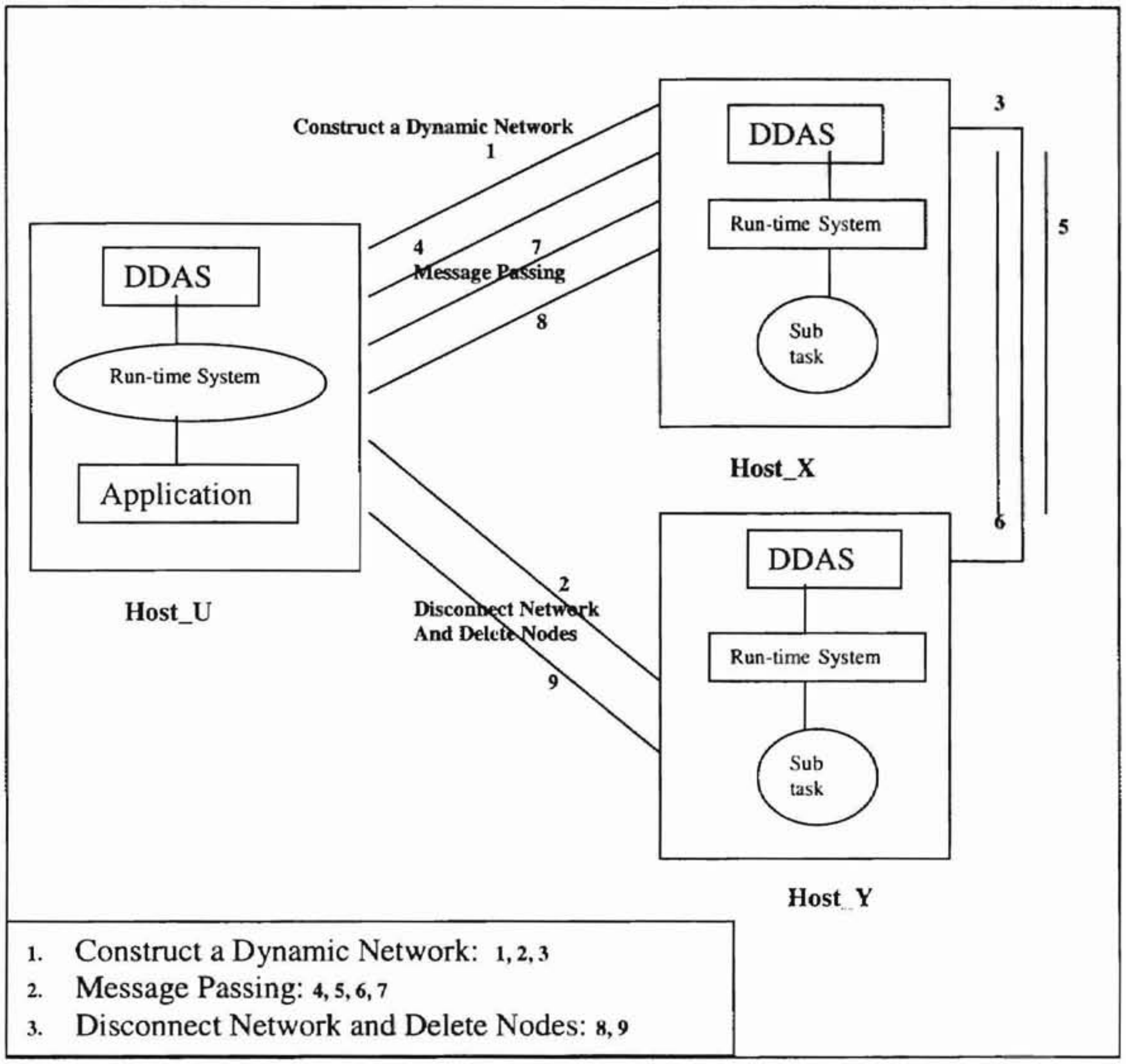
3.3 Illustrative Examples

The concepts are illustrated using simple examples. Assume that there are 3 programs p1, p2 and p3. For this example, let us assume that p1 produces N numbers in a file, p2 adds all the numbers and p3 multiplies all the numbers. Then we can write a program [p3, p2]•p1 to produce the set of numbers and to multiply and add them. The programs p2 and p3 can execute in parallel. The FDCS run-time system facilitates the parallel execution and provides the necessary synchronizations.

As a second example consider the application of merge sort using a 3-node network with hosts U, X and Y. Let q, s and m represent the tasks of sorting (a split segment), splitting and merging respectively. Let one, two and three be functions that select the first, second and third elements from a list respectively. The application can be written as a program P where P = m•[one, m•[two, three]] •(APPLY-TO-ALL q)

•[s•one, two] •s. If P is applied to a list, it is split into two lists. The first list is further divided into two lists. All three lists are sorted using the program q. (The sorting function q can be distributed to 3 nodes of a network and executed concurrently.) The second and third sorted lists are merged, and the result is merged with the first sorted list producing the original list as a sorted list.

The conceptual operating scheme of FDCS using the DDAS system is demonstrated in Figure 8. A 3-node network is assumed for this description. The DDAS systems running on any two machines by the same user can communicate with each other. Therefore, the set of machines forms a fully connected network. Figure 9 shows the mapping of the program P into the network. For the sake of simplicity of description, let us assume that the application program P initiates execution in host U. The first split is done in U. Then the construction functional form is capable of spawning two independent subtasks executable concurrently. Let us assume that the first task (s•one) is assigned to host X and the other one is assigned to U. When the task assigned to X completes there are three lists, one in U and the other two in X. Since the next task is to apply q to all lists, the two lists in X can be sorted simultaneously. So, one of the lists is copied to Y. Now, the sorting function can be applied concurrently to the three lists. They execute in hosts U, X and Y. When the third list is sorted in Y, it is copied to node X where a merge takes place. When the merge is completed, the resulting list is copied to node U where the final merge is done. DDAS may be used to for copying data and code. The DDAS commands can be used to initiate remote task execution and to control them.



1. Construct a Dynamic Network: 1, 2, 3
2. Message Passing: 4, 5, 6, 7
3. Disconnect Network and Delete Nodes: 8, 9

Figure 8. DDAS System (Adopted from [17])

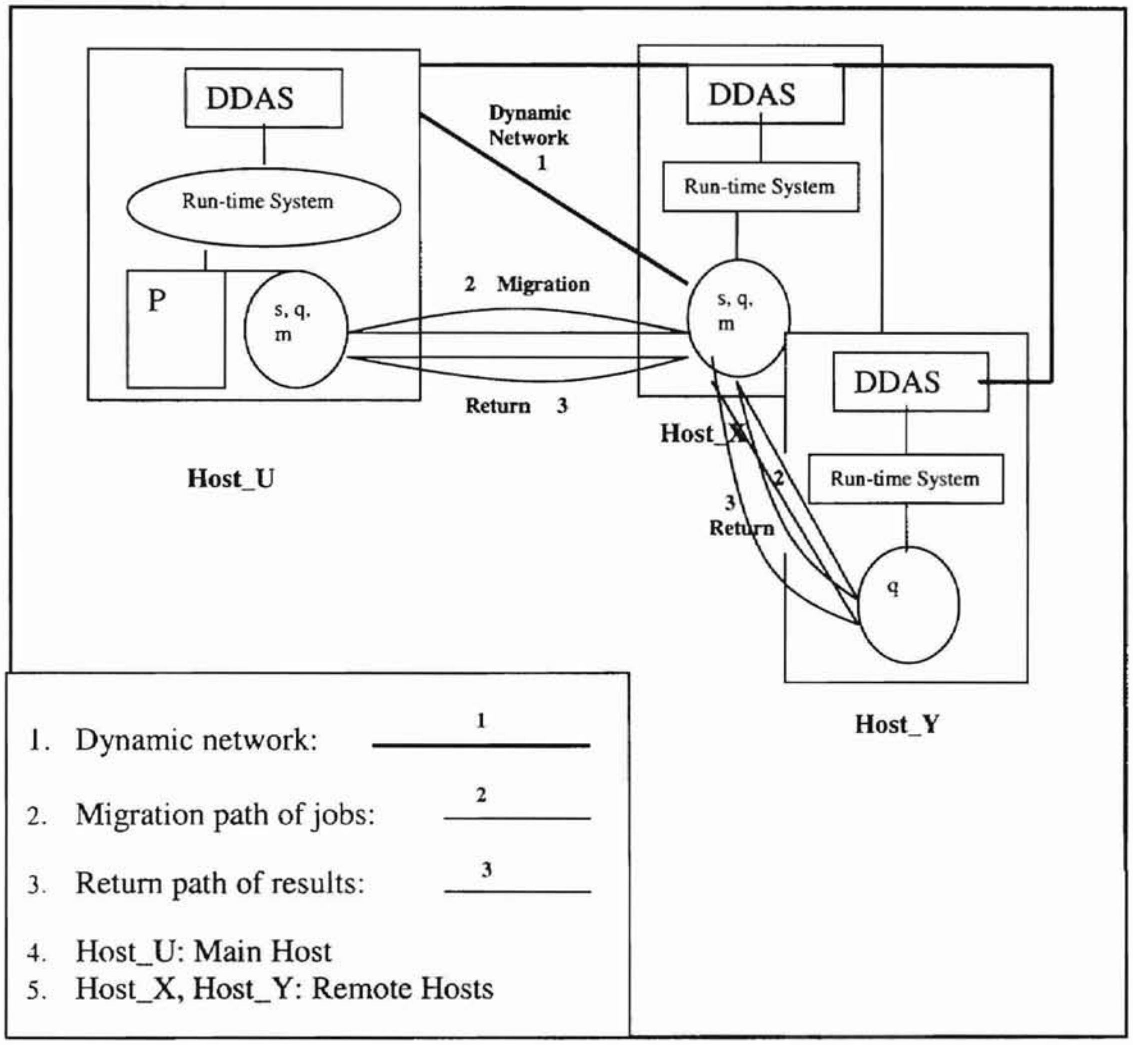


Figure 9. Mapping of an Application to the DDAS Network (Adopted from [30])

In the next chapter, we present the design of a distributed system that supports some of the functional forms.

CHAPTER 4

A DISTRIBUTED SYSTEM DESIGN FOR COMPOSITION AND PIPELINE

The system is modeled as a three-layered architecture. The top-level layer is the application program. The lowest layer is the DDAS kernel, which handles all the communications between nodes of a network. The middle-layer is the run-time system. It provides the mechanisms for coordination and dispatch of independent program units. The three layers are described in the next three sections.

4.1 Top-level Layer — Application

It is a collection of executable programs and compiled to run in specific types of computers in the distributed system. The syntax and semantics of each application is adopted from the FDCS model to composition and pipeline framework.

In this research, all applications are assumed to be built using only one combining form. (Even though this is a limitation, user will still be able to build applications using multiple combining forms manually.)

4.2 Lowest-level Layer — DDAS

DDAS is adopted as the transport system. However, DDAS outlined in [17, 29] is substantially modified in this research. A new Telnet class based on IOtest class [31] is employed. The modifications are highlighted below:

1. By using a new Telnet class and a new FTP configuration, the values, such as user ID, password and the path, which are used to construct the distributed computing system, can be read at run-time. This change makes the coupling loose. The user ID, password, or path may be changed without affecting the system. Thus, the new design enhances scalability and extensibility.
2. The DDAS kernel is designed as a separate package. The DDAS kernel is reconstructed to provide three services used by the run-time system as shown in Figure 10. They are remote execute and delete service (REDS), migration service (MS), and message passing service (MPS). Telnet [31], FTP and synchronization [32] are underlying components in constructing a dynamic network.

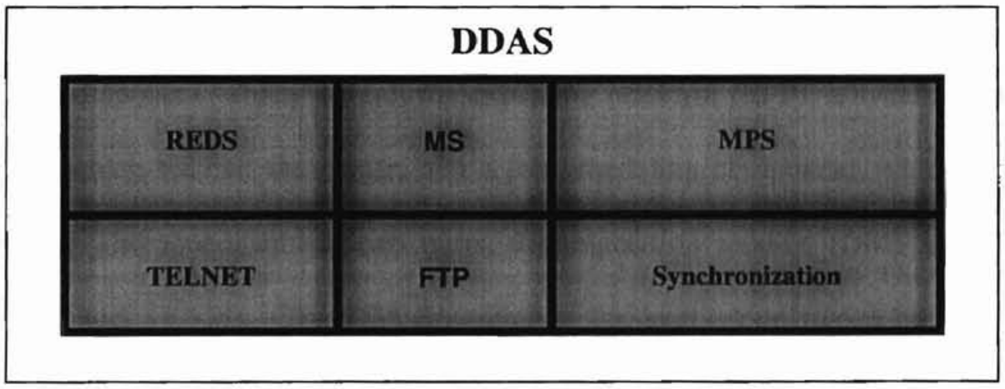


Figure 10. DDAS Configuration

3. The DDAS nodes can be established individually which makes its construction avoid the down nodes when they are detected. Moreover, the remote node can be added only when it is needed. That is, it increases the adaptive capability of the system.

4. A router is designed to provide the run-time routing, which makes the DDAS system more powerful and also saves manual routing.
5. The DDAS input command set is rebuilt. I describe it in detail in the next chapter. For example, when disconnecting a node, the command will guarantee that the remote process is killed before cleaning the node.

4.3 Middle-lever Layer -- Run-time System (RTS)

The run-time system provides the mechanisms for coordination and dispatch of independent program units based on composition and pipeline programming. It serves as a scheduler for all programs, allowing each program unit to execute and communicate with each other based on composition or pipeline organization.

Figure 11 gives an overview of the run-time system. It consists of a job submission component, which we call the executing platform (EPF); a syntax checking component, which we call the parser; and a job scheduling component, which we call "scheduler". The scheduler has two parts, decomposition module (DM) and virtual network manager (VNM). The DM decomposes the application into a collection of executable program units based on the syntax and semantics of the application. It also provides services to spawn each program unit. The VNM based on the DDAS kernel builds the corresponding network and serves as a coordinator and dispatcher of the executable program units. Table 1 illustrates the responsibilities of each component of the run-time system.

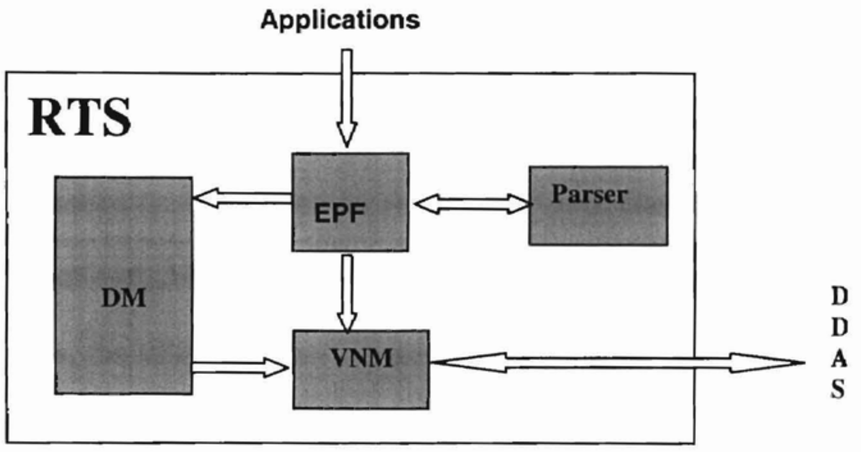


Figure 11. Run-time System Overview

Table 1. Responsibilities of Each Component of RTS

	Responsibilities
EPF	1. Provides a platform to submit applications and commands 2. Communicates with the parser and forwards the application to DM 3. Invokes VNM.
Parser	Checks the syntax of the submitted application
DM	1. Decomposes the application to a collection of executable program units 2. Provides services to spawn each executable program unit
VNM	1. Constructs a distributed system for functional computing 2. Serves as a coordinator and dispatcher of executable program units

Section 4.3.1 describes the basic operations of the RTS. Sections 4.3.2, 4.3.3, 4.3.4 and 4.3.5 describe each component of RTS, that is, EPF, Parser, DM and VNM, respectively.

4.3.1 Basic Operations of the Run-time System

First, the run-time system reads the virtual network setup data file (vns.data) and stores the information in virtual network setup data (Vnsd) class for future use. (vns.data contains the list of available computers.) Then it invokes the EPF to activate the run-time system. That is, the EPF provides a system prompt and waits for the user to input the EPF command. The following is an outline of the basic operations of the RTS as illustrated in Figure 12.

1. The EPF provides a platform to submit applications and commands.
2. If the submission is an application, EPF invokes the Parser.
3. The Parser checks the application.
4. After the Parser finishes its checking, the result is communicated to EPF. If the syntax is correct, the EPF then checks whether the application is executable or not. If the application is executable, it invokes the DM.
5. The DM decomposes the application to a collection of executable program units and provides services to spawn each executable program unit.
6. If the user confirms its execution, VNM is invoked.
7. The VNM constructs a distributed system for executing the application.
8. During the construction, the VNM communicates with DM to get the spawning process. Meanwhile, the VNM coordinates the executable codes and dispatches them.
9. The VNM manages DDAS while it is active.
10. Once the distributed computing finishes, the RTS returns to the EPF execution state.

11. If there are more applications to be computed, the user may continue to submit them until all are done. The exit command terminates the RTS.

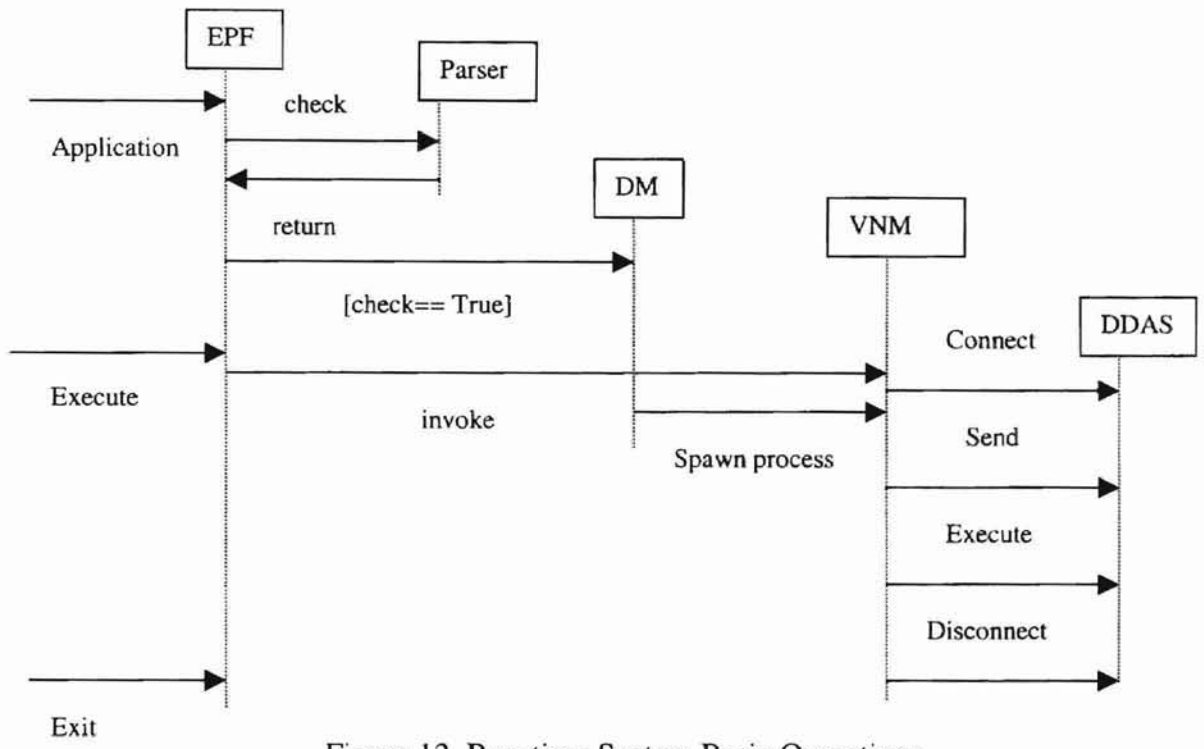


Figure 12. Run-time System Basic Operations

4.3.2 Executing Platform (EPF)

The EPF allows the users to input their application or commands. It communicates with the Parser and forwards the application to the DM. It also invokes the VNM when the user confirms to execute the application.

The EPF commands consist of Application command, Execute command and Exit command. The Application command is used to input the application. The format is:

Application <cftype> <application>

As we mentioned in section 4.1, in this research, all applications are assumed to be built using only one combining form. By this assumption, the <cftype> and

<application> options take arguments, listed in Table 2 and Table 3.

Table 2. <cftype> Options

Option	Meaning
-0	Stands for composition
-1	Stands for pipeline

Table 3. <application> Options

Option	Meaning
<name> [. <name> ...] : <indata>	composition
<name> [|| <name> ...] : <indata>	Pipeline

Naming methods of an application are presented in the next chapter. The reader is referred to it to see the detailed naming policy used in this research.

The Execute command and Exit command do not have any argument. Their formats are "Execute" and "Exit", respectively. The Execute command is used to invoke the VNM, which builds a distributed system for executing the functional form. The Exit command is used to quit the run-time system.

4.3.3 Parser

The Parser is used to determine whether the application is syntactically well formed or not. In this research, non-recursive predictive parsing is employed. The model is shown in Figure 13.

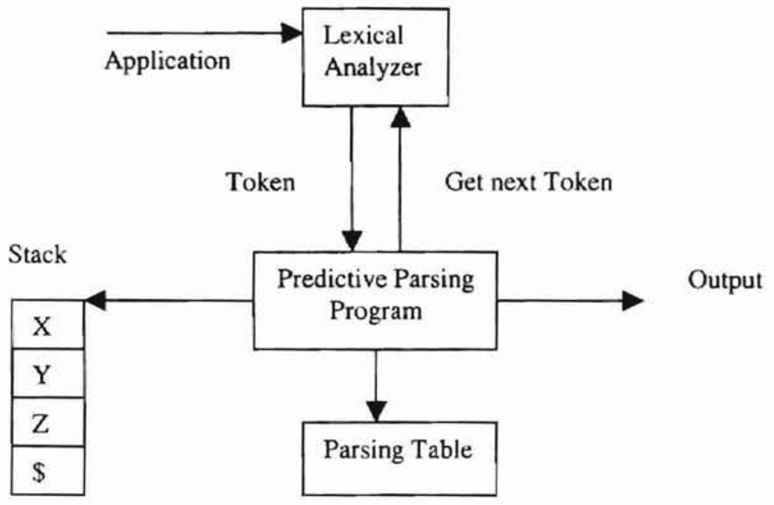


Figure 13. Model of a Non-Recursive Predictive Parser

This non-recursive predictive parser has a lexical analyzer, a stack, a parsing table, and a predictive parsing program. The lexical analyzer is to extract the next token from application and give it to the parsing program. The stack contains a sequence of grammar symbols with $ on the bottom, indicating the bottom of the stack. Initially, the stack contains the start symbol of the grammar on top of $. The parsing table is a two dimensional array M[A,a], where A is a nonterminal, and a is a terminal or the symbol $. The Parser is controlled by the predictive parsing program. The behavior of the parser can be described in terms of its configurations, which give the stack contents and the remaining input.

4.3.4 Decomposition Module (DM)

Before execution, the application should be decomposed to a collection of executable program units. It is accomplished by the decomposition module. The DM also provides operations to spawn processes. Figure 14 shows the decomposition model.

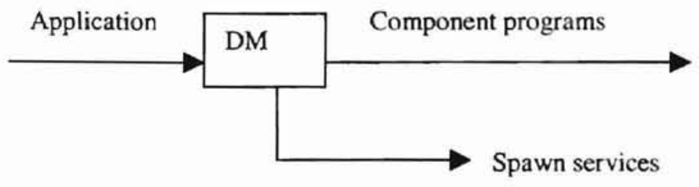


Figure 14. Decomposition Module

The core component of the DM is a decomposer. The decomposer stores the information about each executable program unit. The information should have the following components and the first four components should be an array of size N. N is the number of programs in this application. We call the stored information a meta functional unit (MFU).

Code Name	Code Type(Java, C or C++)	Input	Output	Number of Programs in this application

Figure 15. Basic Components of a MFU

In this design, code name, code type and the first input file can be read from the name of an application itself. The details of naming an application are provided in the next chapter. The output file name will be created by the system automatically. The number of programs contained in this combining form will be detected by the system also.

For each decomposer, it also must provide an operation to spawn the process. That is, it must provide process creation and execution services.

4.3.5 Virtual Network Manager (VNM)

Different combining forms have their own specific execution and communication order rules. The VNM helps to construct a different distributed system for different combining forms. During the construction, the VNM communicates with the DM to get the spawning process and the related information stored in the MFU. It serves as a coordinator and dispatcher for executable program units. It also maintains the DDAS while it is active.

This section describes the basic execution and communication rules of composition and pipeline, respectively, and the role of VNM in each case, illustrates the virtual network, which includes the command set of VNM for constructing and maintaining the DDAS, and presents the VNM environment for pipeline programming.

4.3.5.1 Composition

In composition, the component executable program units of an application are executed in sequential order, which means that the next one can run only after the previous one has terminated. Therefore, a looping structure is used to control the execution order of the codes. In composition, all tasks can be done (1) in one processor one after the other, or (2) can be done in different processors one by one. In this research, we use the first approach since that is the most efficient way for an inherently sequential composition. It reduces the overhead of message passing. We use the second approach as a special case of pipeline, which will be described in the next section. By this assumption, the components in composition do not need to be distributed to different machines. So VNM does not need to build a network for it.

4.3.5.2 Pipeline

In pipeline, the component executable program units of an application are executed in pipeline order, which means that the programs are performed in succession and may run in parallel at the same time. In pipeline, each task will be executed by a separate process or processor, as shown in Figure 16. If each program is performed in succession but not in parallel at the same time, it is the second approach of composition as we mentioned above. Meanwhile, it is also a special case of pipeline. This special case is suitable for workload balancing. When the first node is too busy, while the other nodes are idle, it can distribute tasks to idle nodes. Also, it can serve as barrier synchronization if composition involves functions such as the ones built from construction functional form.

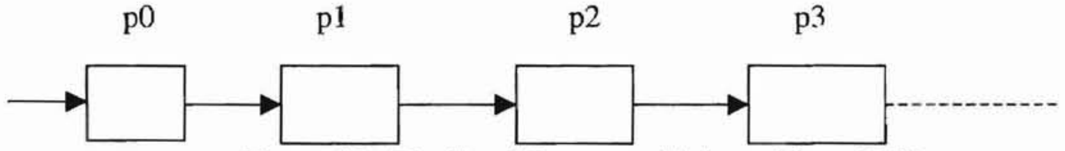


Figure 16. Pipelined Processes (Adopted from [33])

A key requirement for pipeline is that the communication happens only between adjacent processes. This suggests direct communication links between the processors onto which adjacent processes are mapped. An ideal interconnection structure is a line or ring structure such as line of processors connected to a host system as shown in Figure 17 and Figure 18. In this research, a ring structure is employed.

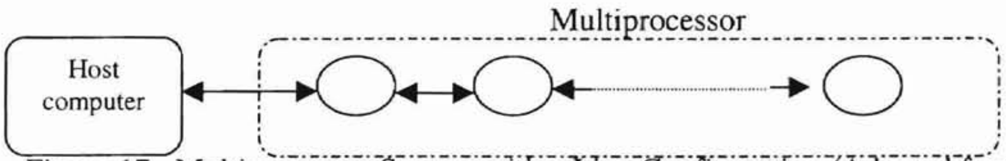


Figure 17. Multiprocessor System with a Line Configuration (Adopted from [33])

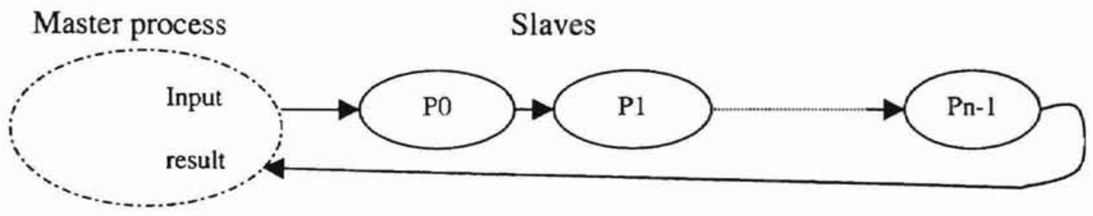


Figure 18. Pipeline with a Master Process and Ring Configuration (Adopted from [33])

4.3.5.3 Virtual Network

The virtual network is a collection of identical processing elements, also called nodes or processors. Though concurrent processors can be composed of a variety of different nodes, in this research, an ensemble of identical nodes is being used. Each of the nodes executes its own instruction stream and operates on a separate set of data. That is, the nodes comprise a MIMD machine structure with no shared memory facilities. All communication between nodes is accomplished by passing messages. In this research, a modified DDAS is employed to provide all communication between nodes.

Having introduced the virtual network that we shall use to explore concurrent programming techniques, we now define the command set which will be used to construct and maintain the virtual network. Table 4 shows the command set provided by the VNM to construct and maintain the virtual network.

Table 4. Command Set of the VNM

Command	Option	Meaning
Connect		1. Send FDCS executable codes to the remote node/s
	Machine	2. Remote execute FDCS components
Disconnect		1. Send message to kill the remote process
	Machine	2. Remote delete FDCS components
Send	machine f1	Send the file f1 to the remote machine
	machine f1 f2	Send the file f1 to the remote machine named as f2
Execute		Invoke program execution

4.3.5.4 VNM Environment for Pipeline Programming

The VNM environment for pipeline programming is based on the following algorithm for a pipeline application (adopted from [35]):

1. Create a set of processors.
2. Assign a task to each processor.
3. The first task performs calculations on a portion of the data, writes the results to a shared file and notifies the next task that the results are available for processing.
4. Add additional tasks, giving the first task new data sets and having each subsequent task use the results of the previous task until all the work is done.
5. When work runs out, each task terminates or, if there are other tasks to be

done, relinquishes its processor or spins until it is assigned a new task.

6. Proceed with serial execution.

Figure 19 illustrates the pipeline programming model.

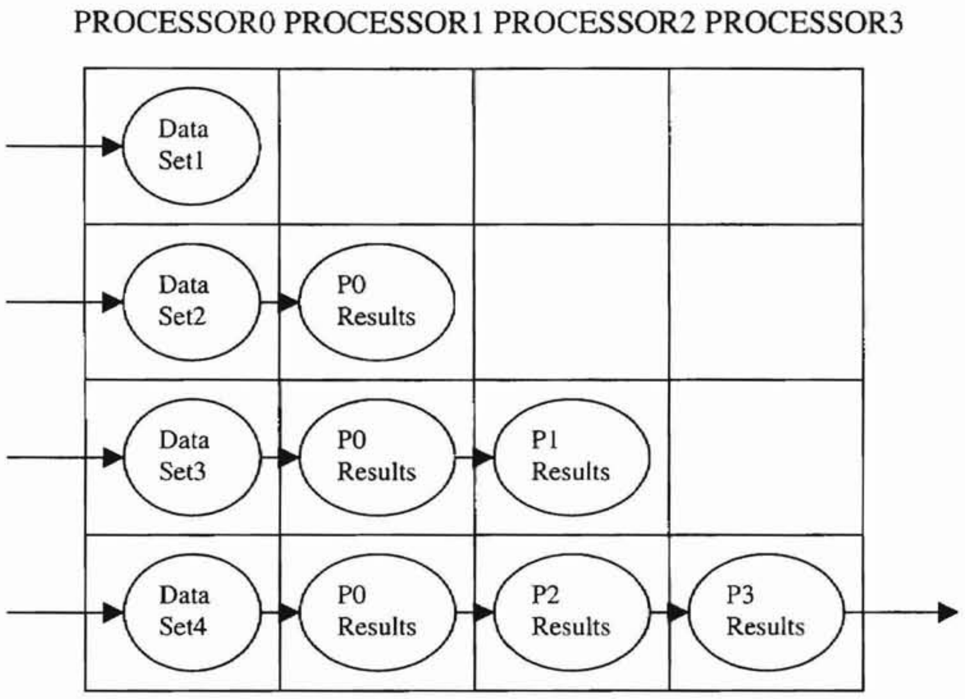
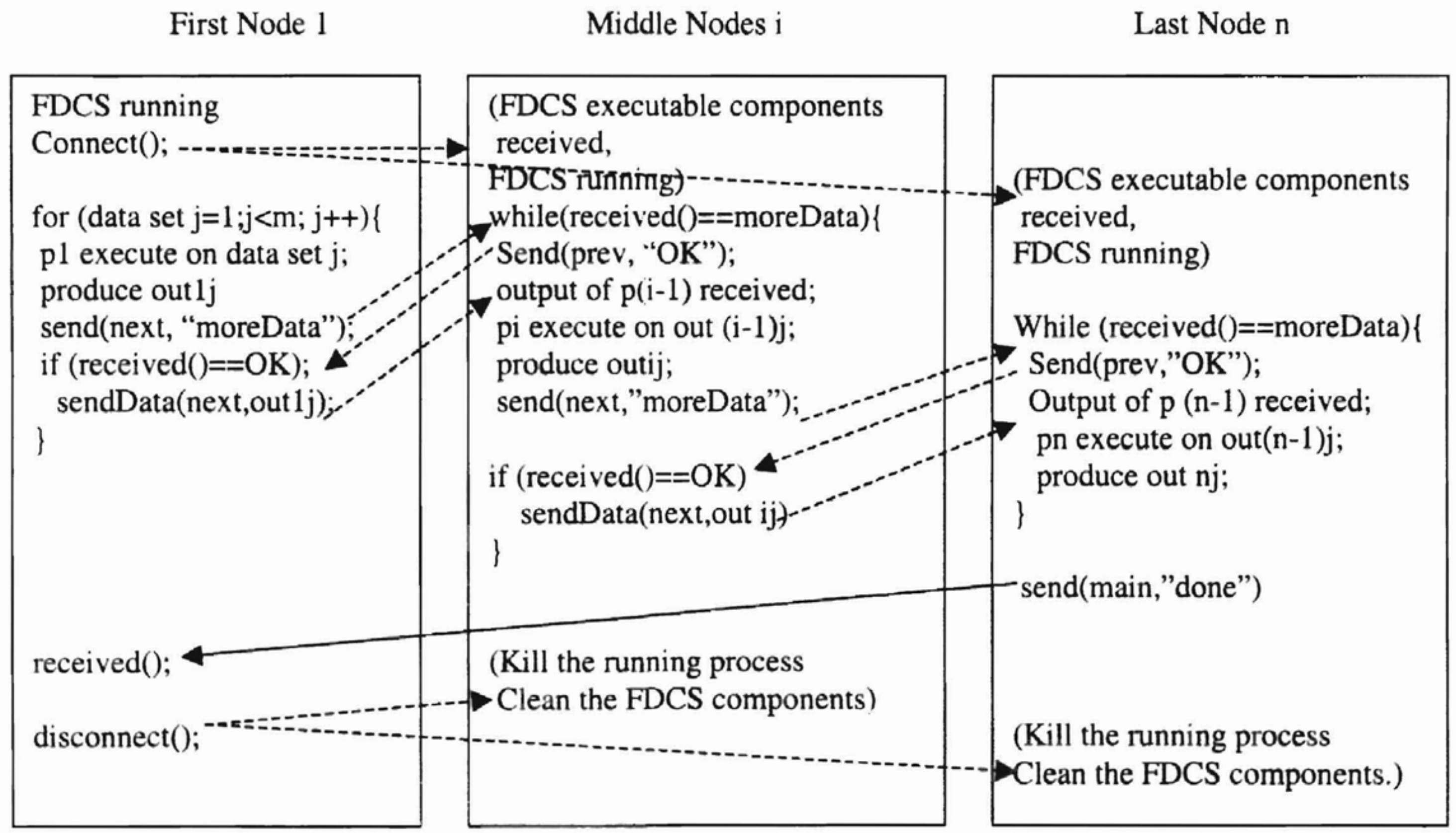


Figure 19. Pipeline Programming Model (Adopted from [35])

Figure 20 illustrates the VNM environment for pipeline programming as implemented in this research.



```
FDCS running
Connect();

for (data set j=1;j<m; j++){
 p1 execute on data set j;
 produce out1j
 send(next, "moreData");
 if (received()==OK);
  sendData(next,out1j);
}

received();

disconnect();
```

```
(FDCS executable components
 received,
FDCS running)
while(received()==moreData){
 Send(prev, "OK");
 output of p(i-1) received;
 pi execute on out (i-1)j;
 produce outij;
 send(next,"moreData");

if (received()==OK)
   sendData(next,out ij)
}

(Kill the running process
 Clean the FDCS components)
```

```
(FDCS executable components
 received,
FDCS running)

While (received()==moreData){
 Send(prev,"OK");
 Output of p (n-1) received;
  pn execute on out(n-1)j;
  produce out nj;
}
 send(main,"done")

(Kill the running process
Clean the FDCS components.)
```

Figure 20. VNM Environment for Pipeline Programming

CHAPTER 5

SYSTEM IMPLEMENTATION

In this chapter, the implementation of the three layers defined in the previous chapter is described. We begin this chapter with a Context Free Grammar specification of the top-level layer. Following the top-level layer, we show in detail the implementation of the lowest-level layer -- DDAS kernel. All the basic routines in DDAS kernel will not refer to any particular variable or to any interconnection topology. It has been implemented to be a totally independent package, which can be easily installed on other machines. Finally, the middle-level layer – RTS is discussed. Even though each component of RTS is discussed, emphasis is placed on the VNM. In addition to the discussion in this chapter, Appendix B provides some source code.

5.1 Top-level Layer – Application

Each application is a collection of executable program units written in Java, C or C++. In this research, as we mentioned in the previous chapter, all applications are assumed to be built using only one combining form. The specification of each application is given below as a Context Free Grammar (CFG). The start symbol of the CFG definition of application is the variable <application>. Non-terminals are specified between < and > symbols.

<application> $\rightarrow$ <program-unit><space> : <space><input-data>

<program-unit> $\rightarrow$ <program> <repeat>

<repeat> $\rightarrow$. <space><program><repeat> (only for composition)

<repeat> $\rightarrow$ || <space><program><repeat> (only for pipeline)

<repeat> $\rightarrow \varepsilon$

<program> $\rightarrow$ <ID>_<exec-type>

<input> $\rightarrow$ <ID>.<data-suffix>

<ID> $\rightarrow$id

<exec-type> $\rightarrow$ j

<exec-type> $\rightarrow$ c

<space> $\rightarrow$ any sequence of one or more blanks

<data-suffix> $\rightarrow$ dat

In this research, each <program> contains <ID>, _, and <exec-type>. By using this specification form for program names, the languages of implementation can be inferred by the RTS.

5.2 Lowest-level Layer – DDAS Kernel

A modified DDAS (as we mentioned in the previous chapter) is implemented in this research. DDAS manager is included in the next section as part of the RTS. In this section, we describe the implementation of the DDAS kernel only. We first describe the implementation of the underlying components. We then describe the new virtual network data set up file (vns.data), which is used to support this modified DDAS kernel. Finally, we describe the role of this DDAS kernel. That is, the service routines provided by the

DDAS kernel.

DDAS kernel is a small set of data structures and routines that are at the core of this distributed system. Telnet, FTP and synchronization are underlying components. A new Telnet class based on IOtest class [31] is shown in Figure 21 and a new FTP file configuration is shown in Figure 22 respectively. The synchronization component is adopted from [32]. Port 23 is used in Telnet, port 21 is used in FTP, and port 7003 is used in synchronization. By these modifications, all the parameters used to construct the distributed computing system can be read at run-time.

```
package DDAS.TELNET;
import java.io. *;
public class Telnet {
   TelnetIO tio = new TelnetIO();
   public Telnet (int function-type, String host, String userId, String passWd, String msg,
    String command, String  msgMore, String runMsg, String changeDir, String makeEx) {
        try {
                tio.connect(host, port);  // connect other host
                wait("login:");           // wait for receiving "login"
                send(userId);             // send userId
                wait("password:");        // wait for receiving "password"
                send(password);           // send password
                wait(msg);                // wait for receiving prompt
                send(changeDir);          // send changing directory
                wait(msgMore);            // wait for receiving more prompt
                if (function-type==EXECUTE)
                   send(makeEx);          // make executable file
                send(command);            // send function command
                if (function-type==EXECUTE)
                  wait(runMsg);
                if (function-type==DELETE)
                  wait(msgMore);
                tio.disconnect();         // disconnect form the host
         } catch (IOException e)
                e.printStackTrace();
   }
   // skip any received data until the checkData appears
    private synchronized void wait(String checkData){
        String receivedData = null;
        do {
              try{
                   receivedData = new String(tio.receive());
                 }catch (IOException e) {
                   e.printStackTrace();
                 }
         }while (receivedData.indexOf(checkData) == -1);
   }
   //send a string to the remote host, since TelnetIO needs a byte buffer
   // we have to convert to string first
   private synchronized void send (String sendData){
        byte[] buf = new byte[sendData.length()];
        buf = sendData.getBytes();
        try{ tio.send(buf);
        } catch (IOException e){
            e.printStackTrace(); }
   }
}
```

Figure 21. Telnet Class Used in the Modified DDAS

```
#!/bin/sh
ftp -n << !EOF!
open $1
binary
user $2 $3
cd $4
put $5 $6
quit
        !EOF!
```

Figure 22. FTP File Configuration Used in the Modified DDAS

In order to support this modified DDAS kernel functions, the virtual network setup data file (vns.data) need to be modified. It should contain the following fields to enable the modified DDAS system to set-up run-time parameters.

1. Host name of each machine which the user has full access authorization
2. User ID
3. Password for this account
4. The initial prompt after the system login
5. Executable program directory
6. Initial running status

The role of this kernel is to provide remote execute and delete services (Figure 23), migration services (Figure 24), and message passing services (Figure 25).

```
// Remote Execute & Delete Services
// Based on Telnet
public static void  remoteExecute(String host, String userId, String passWd, String msg, String command,
                              String msgMore, String runMsg, String changeDir, String makeEx){
        Telnet re = new Telnet ( EXECUTE, host, userId, passWd, msg, command,
                              msgMore, runMsg, changeDir, makeEx);
}

public static void  remoteDelete(String host, String userId, String passWd, String msg, String command,
                              String msgMore, String changeDir){
        Telnet rd = new Telnet ( DELETE, host, userId, passWd, msg, command,
                              msgMore, “”, changeDir, “”);
}
```

Figure 23. Remote Execute & Delete Services

```
// Migration services
// By using Java Runtime.getRuntime().exec() to create migration process
// Parameter comm is based on FTP command.
public static void sendFiles(String comm, String host, String userId, String passWd){
                commands = comm + " " + host + " " + userId + " " +passWd;
                Process sf = Runtime.getRuntime().exec(commands);
                sf.waitFor();
}

public static void sendData(String comm, String host, String userId, String passWd, String path,
                           String sourceFile, String destinationFile){
                commands = comm + " " + host + " " + userId + " " +passWd  + " " + path
                           + " " + sourceFile + " " + destinationFile;
                Process sf = Runtime.getRuntime().exec(commands);
                sf.waitFor();
}
```

Figure 24. Migration Services

```
 // Message Passing Services
 // Based on Synchronization package
 // Message Send  Routine
  public static void send(String host, String msg) {
          setmsgSend(msg);
          setServer(host);
          msgSend();
 }
 // Message Received Routine
 // Set up sever socket to listen to the client.
 // Once accept the client's request, return Extendrendezvous object to communicate with the client.
  public static void received() {
      openSocket();
      while(lock == 0) pause(PAUSE);
  }
  // Message sendReceived Routine
  public static void msgSendReceived(String host, String msg) {
      send(host, msg);  // Client makes a quest to host.
      received();  // Server receives the quest and makes a reply
   }
   // Message sendReceived Routine
   public static void msgSendReceived(String host1, String msg, String host2) {
      send(host1, msg);// Client makes a quest to host1
      setServer(host2);
       received();  // Server receives the quest and makes a reply
   }
   // Message Send Routine, when the host and the msg has been set
   // Open client socket and make a request to server
   // Once the request has been accepted, return Extendrendezvous object to communicate with the server
   public static void msgSend() {
      connectSocket();
      while(lock == 1) pause(PAUSE);
   }
```

Figure 25. Message Passing Services

The Detailed implementation is given in Appendix B.

5.3 Middle-level Layer -- Run-time system (RTS)

This section describes an implementation of each component of the run-time system. Section 5.3.1 and 5.3.2 give the algorithms for the EPF and the Parser, respectively. Section 5.3.3 describes the implementation of the DM. At last, section 5.3.4 discuss the VNM. Since the VNM is the core of the RTS, our focus is on this sub-section.

5.3.1 Executing Platform (EPF)

We have described the responsibilities of the EPF in Chapter 4. The algorithm for EPF is given below:

Algorithm 1. EPF

```
repeat
  provide "EPF>>" prompt
  Let stdin be the standard input and args be the number of the input tokens
  command = the first token of the input;
  if (command.equals( APPLICATION) && (args >1))
      cfType = the second token of the input;
      if (cfType.equals(COMPOSTION) || cfType.equals(PIPELINE))
           According to the Parser class' construct, standard the input
```

```
                    Parser test = new Parser();
                    if (test.check() = true)
                        According to the DM's construct, standard input;
                        if  the application is executable
                            invoke DM;
                else help();
            else if (command.equals(EXECUTE) && (args ==1))
                invoke VNM;
        until command.equals(EXIT)
```

5.3.2 Parser

This Parser is based on deterministic predictive parsing. An LL(1) parse table [36] is constructed and used. The specific algorithm is given below:

Algorithm 2 -- Parser

Input: A string w and a parsing table M for grammar G.

Output: if w is in L(G), a leftmost derivation of w; otherwise, an error indication.

Method:

```
    set ip to point to the first symbol of w$;
    repeat
    let X be the top stack symbol and a the symbol pointed by ip;
     if X is a terminal or $ then
        if X=a then
```

```
                    pop X from the stack and advance ip
          else error()
    else    // X is a nonterminal
          if M[X,a] = X -> Y1Y2...YK then begin
                    pop X from the stack;
                    push YkYk-1...Y1 onto the stack with Y1 on top;
                    output the production X->Y1Y2...Yn
                    end
          else error()
    until X=$  // stack is empty
```

The push down automata (PDA) M is as follows:

M=({q}, { :, ., ||, id, dat}, {<application>, <program-unit>, <repeat>, <program>, <input>, <ID>, <data-suffix>, :, ., ||, id, dat, $}, δ, q, $, { })

The transition table is as follows:

(1) δ (q, ε, <application>) ={(q, <program-unit>:<input-data>);

(2) δ (q, ε, <program-unit>) ={(q, <program><repeat>);

(3) δ (q, ε, <repeat>) ={(q, .<program><repeat>); (only for composition)

(4) δ (q, ε, <repeat>) ={(q, ||<program><repeat>); (only for pipeline)

(5) δ (q, ε, <repeat>) ={(q, ε);

(6) δ (q, ε, <program>) ={(q, id); //This is a little bit different form the way we specified in section 5.1. We consider <ID>_<exec-type> as a whole program ID, which does not need to be broken in this PDA.

(7) δ (q, ε, <input>) ={(q, <ID>.<data-suffix>);

(8) δ (q, ε, <ID>) ={(q, id);

(9) δ (q, ε, <data-suffix>) ={(q, dat);

(10) δ (q, :, :) ={{q, ε)};

(11) δ (q, ., .) ={{q, ε)};

(12) δ (q, ||, ||) ={{q, ε)};

(13) δ (q, id, id) ={{q, ε)};

(14) δ (q, dat, dat) ={{q, ε)};

(15) δ (q, \$, \$) ={{q, ε)};

This corresponding PDA will accept application strings by empty stack. Refer to the Appendix B for the detailed implementation.

5.3.3 Decomposition Module (DM)

In the previous chapter, we described the design of the decomposition module, and indicated that a decomposer is the core component in this DM. The decomposer is designed to store the information about each executable program unit meta functional unit (MFU), rather than the program itself. Figure 26 shows the data components of the MFU. Figure 27 shows the routines used to retrieve information from a MFU. These routines are used by the VNM to retrieve the useful information. Figure 28 shows how to decompose an application into a collection MFUs. At last, Figure 29 shows how the decomposer creates and executes a process. It shows the process to spawn a process.

```
package DM;
import java.io.*;
import java.util.*;
public class Decomposer implements ExecType{
   private String[] prog; // a collection of executable program units used in this application
   private String[] type; // language types set of each executable program units
   private String[][] input; //input data sets
   private String[][] output; // output data sets
   private int count=0;   //number of programs used in this application
   private int dataCount=0;  // number of data sets used in the first program
   private String[] data;  // data sets consumed by the first program
```

Figure 26. The Information Stored in a MFU

```
public String getProg(int index){
      return prog[index];
   }
   public String getType(int index) {
      return type[index];
   }
   public String getInput(int index, int d){
      return input[index][d];
   }
   public String getOutput(int index,int d){
      return output[index][d];
   }
   public int getProgCount(){
      return count;
   }
   public int getDataCount(){
      return dataCount;
   }
```

Figure 27. The Routines to Retrieve Information from a MFU

```
public Decomposer(int i, String[] program, String indata)throws Exception{
     prog=new String[i];
     type=new String[i];
     getData(indata);
     input=new String[i][dataCount];
     output=new String[i][dataCount];
     StringTokenizer str;
     count =i;
     for (int j=0; j<count; j++){
        str = new StringTokenizer(program[j], "_");
        prog[j]=str.nextToken();
        type[j]="_"+str.nextToken();
        if (j==0)
           for (int n=0; n<dataCount; n++)
              input[j][n]=data[n];
        else
           for (int n=0; n<dataCount; n++)
              input[j][n]="out"+(j-1)+n;
        for (int n=0; n<dataCount; n++)
           output[j][n]="out"+j+n;
     }
  }
  private void getData(String indata) throws Exception{
     BufferedReader stdin=
        new BufferedReader(new FileReader(new File(indata)));
     StringTokenizer str=new StringTokenizer(stdin.readLine());
     dataCount=str.countTokens();
     data=new String[dataCount];
     for (int m=0; m<dataCount; m++)
        data[m]=str.nextToken();
  }
```

Figure 28. Construction of MFUs from an Application

```
// By using Java Runtime.getRuntime().exec() function
public void spawn(int index, int d) throws Exception{
     spawn(type[index],prog[index],input[index][d],output[index][d]);
}
private void spawn(String type,String prog, String input, String output) throws Exception {
    String command=null;
    if (type.equals(JAVACODE))
         command="Execj"+" "+prog+" "+input+" "+output;
    else if (type.equals(OTHER))
         command="Execc"+" "+prog+" "+input+" "+output;
    Process ps= Runtime.getRuntime().exec(command);
     ps.waitFor();
 }
```

Figure 29. DM's Spawn Operation

5.3.4 Virtual Network Manager (VNM)

In Chapter 4 we have described the VNM, the core component of the RTS, which builds the corresponding network as necessary and serves as a coordinator and dispatcher of the executable program units. In this section we first describe the implementation of composition functional form which does not need the VNM to build a network for it by using the first approach. We then describe the command set of VNM, to which the pipeline programming environment will be applied. However, before we proceed to discuss the command set of the VNM, we must first describe the virtual network on which the command set is working. Finally, We present in detail a specific implementation of a virtual network environment for pipeline programming.

5.3.4.1 Composition Functional Form Implementation

As we mentioned before, the component programs in composition are executed one after the other. Therefore, a looping structure is used to control the execution order. And by using this approach, all tasks can be done in one processor, which means that the VNM does not need to build a network for it.

```
public void composition() throws Exception{
  Decomposer t0 = new Decomposer(num, program, indata);

  for (int m=0; m<t0.getProgCount(); m++)
       t0.spawn(m,0);
}
```

Figure 30. Implementation of Composition

The routine composition, shown in Figure 30, communicates with the decomposer to get the information and the spawn operation. The method call t0.getProgCount() returns the number of the component programs in a composition application. A for-loop is used here to do the sequential algorithm.

5.3.4.2 Virtual Network Command Set

Before we can discuss the implementation of the command set, we must first examine some of the details of the virtual network that is used in this research. As mentioned previously, the virtual network is an ensemble of independent processing elements communicating with each other by exchanging messages. In this research, we use the modified DDAS to provide message passing exclusively and make no use of any shared memory facilities. Using those modified DDAS kernel services – REDS, MS, and MPS – each node has the ability to allow it to communicate with other nodes in the ensemble. These kernel services are the underlying routines for the VNM to manage the virtual network. The VNM command set is constructed based on them.

In this research, the ensemble is implemented by using an array of the Vnsd. The Vnsd is a class describing the fields in a virtual network setup node or processor. It contains all the fields required by the virtual network run-time setting up. Actually, this class is used to store all the information read from the vns.data file during its run time set up. Another useful field in this class is a unique identifying number, which will be referred to as procId. Since a copy of the same program is typically loaded into each of the nodes of the virtual network, the processor number is very important for referring to a specific node within the ensemble. In this research, we simply number the nodes with consecutive integers starting at 0. That is, we assign its order in this ensemble to each node as its processor Id. By this assumption, each node's procId is identical to its index in the Vnsd array. In addition, the Vnsd class provides the operations to store or retrieve the information on each field. The size of the array is the total number of nodes in this ensemble, which will be referred to as nproc.

We should now be able to discuss the implementation of the commands used to manage DDAS during its run time. This command set is built on the top of the DDAS kernel.

We begin by introducing connect command which provides the capability to establish the virtual network. The implementation is described below:

Step1: Retrieve the information from the Vnsd class

Step2: FTP FDCS executable components to the remote node

Step3: TELNET

Step3.1: Login

Step3.2: Execute FDCS executable components in the remote node

Step3.3: Logout

Step 4: Mark this node is in use.

This command may have one parameter, or no parameter. If it has a parameter, the parameter is the index of the Vnsd array, which is used to retrieve information from the Vnsd class. If there is no parameter, the information for all the nodes will be assumed to be default values. In this implementation, step1 is used to prepare arguments for DDAS kernel service routines. Step2 is accomplished by using the DDAS kernel MS routine.

Step 3 is done by using the DDAS kernel REDS routine. Step 4 is just to inform the virtual network that this node is not free.

We now discuss its complementary command, disconnect, which kills the remote process and then cleans the remote host. The implementation is as follows:

```
Step1: Retrieve the information from the Vnsd class
Step2: Send "Kill" message to the remote host to kill the process
Step3: TELNET
    Step3.1: Login
    Step3.2: Delete FDCS executable components in the remote host
    Step3.3: Logout
Step4: Mark this node is free.
```

Its implementation is similar to that of the connect command. They both have one or no parameter, and need to retrieve information from the Vnsd class to prepare the arguments for DDAS kernel service routines first. The differences are in steps 2, 3 and 4. In disconnect command, step 2 guarantees to kill the process by using DDAS kernel MPS service first, and then step3 is used to clean the FDCS executable components in the remote host by using DDAS kernel REDS service. If no guarantee to kill the remote process, deadlock may occur. At last, step 4 marks the node is free.

The third command we discuss is the send command. It is based on the DDAS kernel MS sendData routine. As its name indicates, the command is used to send the data to the remote host. One of the features we add here is the ability to send data to the remote node only by indicating the destination machine name, instead of all the fields needed by the system run time setting up. It is more flexible to be used in practice. The process of sending data to a node consists of two basic steps:

Step1: Retrieve the information from the Vnsd class

Step2: FTP data file to the remote machine

Same as the above two commands, step1 is used to prepare arguments for the DDAS kernel service routine. Step 2 is done by using DDAS kernel MS service routine. The protocols in Figure 31 illustrate the use of the send command.

```
//Send sourceFile to remote host and name as destinationFile
private static void sendData(String host, String sourceFile, String destinationFile);

//Send sourceFile to remote host
private static void sendData(String host, String sourceFile)
```

Figure 31. Send Command

Having defined a specific implementation of the above three commands, we are ready to proceed with the implementation of the execute command. In this research, the execute command invokes pipeline application execution. I describe it in detail in the next section.

5.3.4.3 Pipeline Functional Form Implementation

The implementation that we consider here will reflect many of the features of the virtual network presented in the previous sections. However, several new features will be added, such as the interconnection topology and routing.

Before we can proceed to the implementation of the pipeline functional form, we must select an interconnection topology for the ensemble. In chapter 4, we saw that a line or ring structure is an ideal interconnection structure for pipeline. In this research, we use a ring topology.

As mentioned previously, a key requirement for pipelining is the ability to send messages between adjacent processes in the pipeline. This suggests direct communication links between the processors onto which adjacent processes are mapped. A convenient way of mapping is for the process ID identical to the processor ID – procId. While such a mapping scheme is based on the assumption that one process maps to one node. Even though the virtual network allows us to discuss environments in which there are many processes per node. However, for concreteness we will assume a one-to-one mapping between processes and nodes. Having a multitasking environment in each node is necessary for some advanced applications, but for the applications we shall be considering, such an environment would only add unneeded complexity.

Due to the synchronization imposed to the pipeline programming, they are most useful in applications whose communications can be expected by the participating nodes. In addition, any message-forwarding must be done explicitly, requiring that intermediate nodes also "know" in advance about the intended communication. Therefore, routing is a prerequisite.

In this research, the most important thing in run time routing is to let the current remote node know its own machine name. After that, according to the information stored in the Vnsd (virtual network setup data) class, get the previous machine and the next machine. (This is based on the above mapping scheme – the process ID identical to the processor ID.) Figure 32 shows the methods.

We are now ready to describe the pipeline implementation in this virtual network environment. We will examine a sample problem that can be pipelined.

```
static void findCurrMachine(){
     Socket theSocket;
     main=vnsd[0].getmachineName();
      theSocket = new Socket(main,PORT);
      curr=theSocket.getLocalAddress().getHostName();
      if (theSocket!=null)
           theSocket.close();
}

static void routing(){
     int index=-1;
     index=getIndex(curr);
     if (index==0 )
        prev=main;
     else if (index>0)
        prev=vnsd[index-1].getmachineName();

     if (nproc == (index+1))
        next="final";
     else if (nproc>(index+1)){
        next=vnsd[index+1].getmachineName();
     }
  }
```

Figure 32. Run time Routing Method

For our example (adopted from [33]), consider the problem of adding a list of numbers. A pipeline solution could have each process in the pipeline add one number to an accumulation sum, as shown in Figure 33, when one number is held in each process. The partial sum is passed from one process to the next, each process adding its number to the accumulating sum.

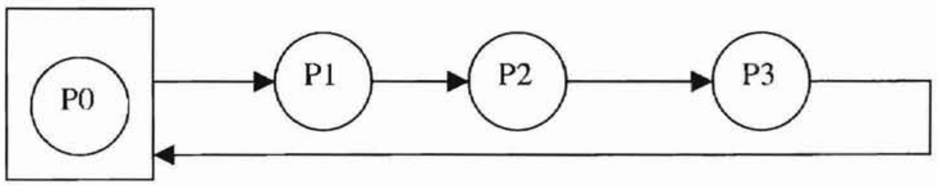


Figure 33. Pipelined Addition (A Ring Architecture)

The basic code for process Pi is simply shown in Figure 34 except for the first process, P0, which is shown in Figure 35; and the last process, Pn-1, which is shown in Figure 36.

```
received();
t0.spawn(getIndex(curr),0);
sendData(next, t0.getOutput(getIndex(curr),0));
send(next, "execute");
```

Figure 34. The Basic Code for the Middle Process Pi

```
t0.spawn(getIndex(curr),0);
sendData(next, t0.getOutput(getIndex(curr),0));
send(next, "execute");
```

Figure 35. The Basic Code for the First Process P1

```
received();
t0.spawn(getIndex(curr),0);
sendData(main, t0.getOutput(getIndex(cur),0));
```

Figure 36. The Basic Code for the Last Process Pn

5.4 Test Environment and Application Examples

In this research, the applications are stored in the main host machine. The set of computers used on an application first must be defined prior to running the programs. This set forms the virtual distributed system. The way of doing this is by creating a list of the names of the computers available in a virtual network set up data (vns.data) file. For these computers, the user must have full privilege. The file is then read by FDCS run-time system. The system used in this research is shown in Table 5.

Table 5. The Domains Used in this Research

Domain	IP Address
a.cs.okstate.edu	139.78.113.1
eslabsvr.wslab.okstate.edu	139.78.113.102
chester.cs.okstate.edu	139.78.96.1

All the applications written in composition and pipeline programming framework can execute in this system.

Examples:

Application written in composition framework as:

```
P1_j . p2_c . p3_c . p4_j : d1.dat
```

Application written in pipeline framework as:

```
SumInt3_j  || SumInt2_c || SumInt1_j  || SumInt_j : d1.dat
```

CHAPTER 6

CONCLUSION AND FUTURE WORK

In this thesis we implement a distributed system environment to execute applications built by composition and pipeline functional forms. We put forth an architectural support for promoting software reusability. Moreover, it provides an environment, which introduces some simultaneity after a complex, time-consuming task has been broken into a series of subtasks. We approach this problem by proposing a three-layered architecture. The lowest layer is the Dynamic Distributed Adaptive System (DDAS), which handles all the communications between nodes of a network. The top-level layer is the application layer, which is a collection of executable programs. The middle-layer is based on the combining forms of composition and pipeline, which provides the run-time mechanisms for coordination and dispatch of independent program units.

We also developed a modified DDAS system to support this system. This modified system has the following new features:

1. Enhance scalability and extensibility of the system
2. Service functions more clear and independent
3. Increase the adaptive capability of the system
4. Provide a run-time routing for pipeline programming
5. Dead lock free

However, due to the time limitation, the following aspect has not been done yet, which will be left as future work:

A distributed system environment based on all functional forms adopted in the FDCS model

BIBLIOGRAPHY

[1] Silberschatz, A. and Galvin, P.B., *Operating System Concepts*, Addison-wesley, Reading, MA, 1998.

[2] Feit, S., TCP/IP: *Architecture, Protocols, and Implementations with Ipv6 and IP Security*, 2nd edition. McGraw-Hill, New York, NY, 1996.

[3] Comer, D.E., *Internetworking with TCP/IP Volume I: Principles, Protocols, and Architecture*, Second Edition, Prentice Hall, Englewood Cliffs, NJ, 1991.

[4] Comer, D.E. and Stevens, D.L., *Internetworking with TCP/IP Volume II: Design, Implementation, and Internals*, Prentice Hall, Englewood Cliffs, NJ, 1991.

[5] Sashidhar, C. and Shatz, S.M., Design and Implementation Issues for Supporting Callback Procedures in RPC_Based Distributed Software, Computer Software and Application Conference, 1997.

[6] Hrivnac, J., *DCE Distributed Computing Environment*, http://hp18.fzu.cz/~hrivnac/DCE/WhatIsIt/index.htm

[7] Farley, J., *Java Distributed Computing*, O'Reilly & Associates, Inc., Sebastopol, CA, 1998.

[8] Java Remote Method Invocation, http://java.sun.com/products/jdk/1.2/docs/guide/rmi/spec/rmiTOC.doc.html

[9] Berson, A., *Client-server Architecture*, 2nd edition, McGraw-Hill, New York, NY, 1996.

[10] Berghoff, J., Drobnik, O., Lingnau, A. and Monch, C., Agent-based configuration management of distributed applications, *Configurable Distributed Systems*, May 1996,

pp. 52-59, IEEE Computer Society.

[11] Gray, R., Kotz, D., Nog, S., Rus, D. and Cybenko, G., Mobile Agents: The Next Generation in Distributed Computing, *Parallel Algorithms/Architecture Synthesis*, March 1997, pp. 8-24, IEEE computer Society.

[12] Ku, H., Luderer, G.W.R. and Subbiah, B., An Intelligent Mobile Agent Framework for Distributed Network Management, Global Telecommunications Conference, Nov. 1997, Vol. 1, pp. 160-164, IEEE.

[13] Evers, X., Jongh, J.F.C.M. de, Boontje, R., Epema D.H.J and Dantzig, R.V., Condor flocking: Load sharing between pools of workstations, *Report 93-104*, The Faculty of Technical mathematics and Informatics, Delft, The Netherlands.

[14] Epema, D.H.J., Livny, M., Dantzig, R.V., Evers, X. and Pruyne, J., A worldwide flock of Condor: Load sharing among workstation clusters, *Report 95-130*, The Faculty of Technical Mathematics and Informatics, Delft, The Netherlands.

[15] Carriero, N., Freeman, E., Gelerner, D. and Kaminsky, D., Adaptive Parallelism and Piranha, *Computer*, Vol. 28, Issue 1, pp. 40-49, IEEE Computer Society.

[16] Gelernter, D. and Kaminsky, D., Supercomputing out of recycled garbage: preliminary experience with Piranha, *Proceedings of the 1992 international conference on Supercomputing*, pp. 417-427, ACM, Inc.

[17] George, K.M. and Kim, K.S., An Adaptive Distributed Computation Support System, International Conference on Parallel and Distributed Processing Techniques and Applications, 1999.

[18] Backus, J., Can Programming be Liberated From the Von Neumann Style? A Functional Style and Its Algebra of Programs, Communicational of the ACM, August

1978, pp 613-641.

[19] Nelson, B.J., *Remote Procedure Call*, XEROX PARC CSL-81-9, May 1981.

[20] Carpenter, B.E., Cailiau, R., *Experience with Remote Procedure Calls in a Real-time Control System*, Software Practice and Experience, Vol 14(9), 901-907 (Sep 84).

[21] Well, E.J., LAN/WAN integration: internetworking + application interoperability, *Military Communications Conference*, Oct. 1994, Vol. 1, pp. 220-226, IEEE.

[22] Benda, M., Middleware: any client, any server, *IEEE Internet Computing*, July-Aug. 1997, Vol. 1, Issue 4, pp. 94-96, IEEE Computer Society.

[23] Peterson, M.T., DCE: A Guide to Developing Portable Applications, McGraw-Hill, New York, NY, 1995.

[24] Fatah, I.A. and Majumdar, S., Performance Comparison of Architectures for Client-Server Interactions in CORBA, *Distributed Computing Systems*, May 1998, pp. 2-11, IEEE Computer Society.

[25] Wollrath, A., Waldo, J. and Riggs, R., Java-Centric Distributed Computing, IEEE Micro, May-June 1997, Vol. 17, Issue 3, pp. 44-53, IEEE.

[26] Java Remote Method Invocation, http://java.sun.com/products/jdk/1.2/docs/guide/rmi/spec/rmiTOC.doc.html

[27] Online Document, *Mobil Agents: Are they a good idea?* http://www.eit.com/goodies/list/www.lists/www-talk.1995ql/0764.html, 1995.

[28] Berghoff, J., et al., Agent-based Configuration Management of Distributed Applications, Configurable Distributed Systems, 1996.

[29] Kim, K.S. and George, K.M., A Secure Environment For Distributed Computing, International Conference on Parallel and Distributed Processing Techniques and

Applications, 1999.

[30] George, K.M., A Distributed Computing Model Based On FP, International Conference on Parallel and Distributed Processing Techniques and Applications, 2000.

[31] The Java Telnet Applet

http://www.first.gmd.de/persons/leo/java/Telnet

[32] Hartley, S.J., *Concurrent Programming: The Java Programming Language,* Oxford University Press, Oxford, NY, 1998.

[33] Wilkinson, B. and Allen, M., Parallel Programming: *Techniques and applications using Networked workstations and parallel computers*, Prentice Hall, Upper Saddle River, NJ, 1999.

[34] Andrews, G.R., Foundations of Multithreaded, Parallel, and Distributed Programming, Addison-wesley, Reading, MA, 2000.

[35] Osterhaug, A., Guide to Parallel Programming: on Sequent Computer Systems, Prentice Hall, Englewood Cliffs, NJ, 1989.

[36] Aho, A.V., Sethi, R. and Ullman, J.D., Compilers: *Principles, Techniques, and Tools*, Addison-wesley, Reading, MA, 1988.

APPDENDIX A

ABBREVIATIONS AND ACRONYMS

API	Application Programming Interface
CF	Combining Forms
CFG	Context Free Grammar
CORBA	Common Object Request Broker Adapter
DCE	Distributed Computing Environment
DDAS	Dynamic Distributed Adaptive System
DM	Decomposition Module
DOD	Department of Defense
EPF	Executing Platform
FDCS	Functional Distributed Computing System
FTP	File Transfer Protocol
IDL	Interface Description Language
IP	Internet Protocol
IIOP	Internet Inter-ORB Protocol
ISO	International Standardization Organization
LAN	Local Area Network
MFU	Meta Functional Unit

MIMD	Multiple Instruction Multiple Data
OMG	Object Management Group
OSF	Open Software Foundation
OSI	Open Systems Interconnection
RMI	Remote Method Invocation
RPC	Remote Procedure Call
RTS	Run Time System
SQL	Structured Query Language
TCP	Transmission Control Protocol
VN	Virtual Network
VNM	Virtual Network Manager
WAN	Wide Area Network

APPDENDIX B

PARTIAL SYSTEM SOURCE CODE

```
package PARSER;
import java.io.*;
import java.util.*;

/*****************************************************************
 * This Parser is based on deterministic predictive parsing.  An LL(1) parse table
 *  is constructed and used.  The specific algorithm is given below:
 * input: A string w and a parsing table M for grammar G.
 * output: if w is in L(G), a leftmost derivation of w;
 *       otherwise, an error indication.
 * method:
 *      set ip to point to the first symbol of w$;
 *      repeat
 *       let X be the top stack symbol and a the symbol pointed by ip;
 *       if X is a terminal or $ then
 *          if X=a then
 *               pop X from the stack and advance ip
 *          else error()
 *       else     // X is a nonterminal
 *          if M[X,a] = X -> Y1Y2...YK then begin
 *               pop X from the stack;
 *               push YkYk-1...Y1 onto the stack with Y1 on top;
*               output the production X->Y1Y2...Yn
 *          end
 *          else error()
 *      until X=$  // stack is empty
 *****************************************************************/

public class Parser implements CfType{
  private boolean correct=false;
  private PDA pda;
  private int X;
  private int ip;
  private int rule;

  public Parser(String comm, String type){
    pda=new PDA(comm, type);

    while (!(((String)pda.getStack().top()).equals("$"))){

      ip=pda.getIndexOfToken();
      X=pda.getStackTopIndex();
      if (X==-1)  {// terminal symbols
        if (pda.setStack(pda.getFirstToken())) //pop terminal from stack
          pda.removeFirstToken();  //move the pointer which point to the comm
        else
```

```
            error();
        }
        else {  // nonterminal sysbols
          String test2=(String) pda.getStack().top();
          rule= parsing_table.M[X][ip];
          if (rule!=-1){   //there is a production
            pda.setStack(rule);  // push the production into the stack
          }
          else error();
        }

      }
      correct=true;
    }

    private void error(){
      System.out.println("Invalid command.");
      System.exit(0);
    }

    public boolean check(){
      return correct;
    }
}

/************************************************************
The PDA is constructed based on the CFG.
The CFG is as follows:
G=({S, E, K, X, P}, {., ||, :, program, input}, Productions, S),
where "program" means the input executable file name and "input"
means the input data file name. The productions are defined
as follows:
 1) S->E:X
 2) E->PK
 31) K->.PK (only for composition)
 32)K->||PK  (only for pipeline)
 4) K->ε
 5) X->input
 6) P->program

The PDA M is as follows:
M=({q}, {:, ., ||, program, input}, {S, E, K, X, P, :, ., ||,program, input, $ }, δ, q, $, {})
The transition table is as follows:
(1) δ (q, ε, S)={(q, E:X)};
(2) δ (q, ε, E)={(q, PK)};
(31) δ (q, ε, K)={(q, .PK)};
```

```
(31) δ (q, ε, K)={(q, ||PK)};
(4) δ (q, ε, K)={(q, ε)};
(5) δ (q, ε, X)={(q, input)};
(6) δ (q, ε, P)={(q, program)};
(7) δ (q, :, :)={(q, ε)};
(81) δ (q, ., .)={(q, ε)}; (only for composition)
(82) δ (q, ||, ||)={(q, ε)}; (only for pipeline)
(9) δ (q, input, input)={(q, ε)};
(10) δ (q, program, program)={(q, ε)};
(11) δ (q, $, $ )={(q, ε)};

This corresponding PDA will accept strings by Empty stack.
*******************************************************************/
class PDA implements CfType{
  private STACK stack; //PDA stack
  private String pdaexp; //expression accepted/rejected by the PDA
  private Token[] tk=new Token[MAX_TOKEN_SIZE];
  private int count=0;
  private int ip=0; // token index pointer;
  private int type;

  public PDA(String comm, String typ){
    stack = new STACK(); // Create a new stack, initially it is empty
    stack.push("$"); // Push start symbol "S" in stack
    stack.push("S"); // Push "$" symbol in stack
    pdaexp=comm+" $"; // Add "$" symbol at the end of the command
    type=Integer.valueOf(typ.substring(1)).intValue();
    setTokenArray();
  }

  private void setTokenArray() throws StringIndexOutOfBoundsException{
    StringTokenizer str=new StringTokenizer(pdaexp);
    count=str.countTokens(); // real number of tokens
    for (int i=0; i< count; i++){
      String temp=str.nextToken();
      tk[i]=new Token(temp,type);
    }
  }

  public String getCommand(){
    return pdaexp;
  }

  public STACK getStack(){
    return stack;
  }
```

```
public String getFirstToken() throws StringIndexOutOfBoundsException{
  return tk[ip].getToken();   //ip always points to the first token of the expression
}
public int getIndexOfToken() throws StringIndexOutOfBoundsException{
  String first=getFirstToken();
  return tk[ip].getIndex();
}

public void removeFirstToken() throws StringIndexOutOfBoundsException{
  ip ++;
}

public int getStackTopIndex(){
  if (stack.top()=="S")
    return 0;
  else if (stack.top()=="E")
    return 1;
  else if (stack.top()=="K")
    return 2;
  else if (stack.top()=="X")
    return 3;
  else if (stack.top()=="P")
    return 4;
  return -1; //terminal symbols
}

public void setStack(int production){
  switch(production){
  case 1: stack.pop(); stack.push("X"); stack.push(":"); stack.push("E"); break;
  case 2: stack.pop(); stack.push("K"); stack.push("P"); break;
  case 3: stack.pop(); stack.push("K"); stack.push("P");
    if (type==COMP){
      stack.push(".");
    }
    else if (type==PIPE){
      stack.push("||");
    }
    break;
  case 4: stack.pop(); break;
  case 5: stack.pop(); stack.push("input");  break;
  case 6: stack.pop(); stack.push("program");  break;
  case -1:// System.out.println("case -1: Grammar error"); break;
  default: break;
  }
}
```

```
private void printStack(int production){
   System.out.println("type="+type);
   switch(production){
   case 1: System.out.println("pop: S   push: E:X"); break;
   case 2: System.out.println("pop: E   push: PK"); break;
   case 3: if (type==COMP)
            System.out.println("pop: K   push: .PK");
            System.out.println("pop: K   push: .PK");
         else if (type==PIPE)
             System.out.println("pop: K   push: ||PK");
         break;
   case 4: System.out.println("pop: K   "); break;
   case 5: System.out.println("pop: X   push: input"); break;
   case 6: System.out.println("pop: P   push: program"); break;
   case -1: System.out.println("case -1: no production"); break;
   default: break;
   }
}

public boolean setStack(String str){
   if ((stack.top()==":") && (str.equals(":"))){
      stack.pop();
      return true;
   }
   else if (((stack.top()==".") && (str.equals("."))&&(type==COMP))
         ||((stack.top()=="||")&&(str.equals("||"))&&(type==PIPE))){
      stack.pop();
      return true;
   }
   else if ((stack.top()=="$") && (str.equals("$"))){
      stack.pop();
      return true;
   }
   else if (stack.top()=="program"){
      Token check=new Token(str,type);
      if (check.getIndex()==2){
         stack.pop();
         return true;
      }
      else return false;
   }
   else if (stack.top()=="input"){
      Token check=new Token(str,type);
      if (check.getIndex()==3){
         stack.pop();
         return true;
```

```
            }
            else return false;
        }
        return false;
    }
}

class STACK extends Stack{
    private static int count;

    public STACK(){
        super();
        count=0;
    }

    public Object top() throws EmptyStackException{
        return peek();
    }

    public void incrementCount(){
        count++;
    }

    public void decrementCount(){
        count--;
    }

    public int getCount(){
        return count;
    }
}

/************************************************************************
 * LL(1) parse table M[A,a]: A stands for Nonterminal symbols, a stands for the grammar
 * symbols.  A->a. The value of the table M[][]: -1 stands for error and each number
 *stands for the number of the production which is going to be used.
 ************************************************************************/
class parsing_table{
    public static int[][] M=
        //     : ||  program input  $
        //     :  .   program input  $
            {{ -1, -1,   1,     -1, -1}, // S
             {  2,  2,   2,     -1, -1}, // E
             {  4,  3,  -1,     -1, -1}, // K
             { -1, -1,  -1,      5, -1}, // X
             { -1, -1,   6,     -1, -1}, // P
        };
}

class Token implements CfType{
   private String token;
   private int index;

   public int getIndex(){
      return index;
   }

   public String getToken(){
      return token;
   }

   public Token(String token, int type){
      this.token=token;
      if (token.equals(":"))
         index=0;
      else if (token.equals(".")&& (type==COMP))
         index=1;
      else if (token.equals("$"))
         index=4;
      else if (isInput(token))
         index=3;
      else if (isProgram(token))
         index=2;
     else if (token.equals("||")&&(type==PIPE))
         index=1;

   }


   public void printToken(){
      System.out.println("token="+token);
      System.out.println("index="+index);
   }

   private boolean isProgram(String token){
      boolean result=true;

      if (isInput(token)||token.equals("||"))
         result=false;
      return result;
   }
```

```
  private boolean isInput(String token) throws StringIndexOutOfBoundsException
{
     boolean result=false;
     int index=token.indexOf(".dat");
     if (index==-1||index==0)
        return false;
     else
        if (token.substring(token.indexOf(".dat")).equals(".dat"))
           result=true;
     return result;
  }
}

public interface CfType {
  final String COMPOSITION="-0";
  final String PIPELINE="-1";
  final int COMP = 0;
  final int PIPE = 1;
  final int MAX_TOKEN_SIZE = 100;
}
```

```
package DDAS;
/****************************************
*                    Ddas Class
* Function:
* 1. Install Server socket on port 7003
* 2. Connect to the Server socket
* 3. Running Receiving thread
* 4. Running Sending thread
****************************************/
import java.net.*;
import java.io.*;
import DDAS.Synchronization.*;
import DDAS.TELNET.*;

public class Ddas extends Concurrent {
  private static String Server = null;
  private static String msgReceived = null;
  private static String msgSend = null;
  private static int lock = 0;
  protected static final int PAUSE = 50;
  private static final int PORT = 7003;
  public static boolean debug = true;
  public static void setServer(String host){
    Server = host;
  }

  public static void setmsgSend(String msg){
    msgSend= msg;
  }

  public static void setmsgReceived(String received){
    msgReceived= received;
  }

  public static void setLock(int myLock){
    lock= myLock;
  }

  public static String getServer(){
    return Server;
  }

  public static String getmsgSend(){
    return msgSend;
  }
```

```
 public static String getmsgReceived(){
   return msgReceived;
 }

 public static int getLock(){
   return lock;
 }

 /****************************
  * Message Passing Services      *
  ****************************/
 public static void send(String host, String msg){
   setmsgSend(msg);
   setServer(host);
   msgSend();
 }

 public static void received() {
   openSocket();
   while(lock == 0) pause(PAUSE);
 }

 public static void msgSendReceived(String host, String msg) {
   send(host, msg);   // Client makes a quest.
   received(); // Server receives the quest and makes a reply
 }

 public static void msgSendReceived(String host1, String msg, String host2) {
   send(host1, msg); // Client makes a quest.
   setServer(host2);
   received(); // Server receives the quest and makes a reply
 }

 public static void msgSend() {
   connectSocket();
   while(lock == 1) pause(PAUSE);
 }

// set-up socket as a server
private static void openSocket() {
  lock = 0;

  EstablishRendezvous er = null;
  try {
    er = new EstablishRendezvous(PORT); // open server socket
  } catch (MessagePassingException e) {
      System.err.println(e);
      System.exit(1);
   }

   // install Server socket
    Rendezvous r = null;
    try {
      r = er.serverToClient(); //Server is waiting for client's request and also
          //return ExtendedRendezvous object to communicate with the client
    } catch (MessagePassingException e) {
      System.err.println("Server:" + e);
   }
   // Running Server
    new Receiver(r);
    er.close();
 }

// connect to the Server socket
private static void connectSocket() {
   lock = 1;
   Rendezvous r = null;

   // create a rendezvous to the Server object
   EstablishRendezvous er = null;
   try {
     er = new EstablishRendezvous(Server, PORT);
                                             //client side, save machine name and port
     r = er.clientToServer();                //open client socket and also return
                      // ExtendedRendezvous object to communicate with the server
   } catch (MessagePassingException e) {
     System.err.println(e);
     System.exit(1);
   }
   // running Client
   new Sender(r);
 }

 /**********************************
 * Remote Execute & Delete Services *
 **********************************/
 //1. Remote execute service
public static void remoteExecute(String host, String userId, String passWord,
                                 String msg, String command, String msgMore,
                                 String runMsg, String changeDir, String makeEx) {
   Telnet re=new Telnet(1, host, userId, passWord, msg, command,
                        msgMore, runMsg, changeDir, makeEx);
```

```
}

 //2. Remote delete service
public static void remoteDelete(String host, String userId, String passWord,
                                String msg, String command, String msgMore,
                                String changeDir){
  Telnet rd=new Telnet(0, host, userId, passWord, msg, command, msgMore,"",
                        changeDir, "");
}

/**********************
 * Migration Services: *
 **********************/
//1. Send files to remote machine
public static void sendFiles(String comm, String host, String userId,
                    String passWord, String changeDir){
 String commands = null;
 try {
   commands = comm + " " + host + " " + userId +" "+ passWord +" "
          + changeDir;
   Process pc =
    Runtime.getRuntime().exec(commands);
   pc.waitFor();
 } catch (IOException e) {
  System.err.println(e);
  System.err.println(e);
 } catch (InterruptedException e) {
  System.out.println("");
 }
}

//2. Send data to remote machine
public static void sendData(String comm, String host, String userId, String passWord,
                             String path, String sourceFile, String destinationFile) {
  String commands = null;
  try {
   commands = comm + " " + host + " " + userId+ " " + password + " " +
     path + " " + sourceFile  +  " " + destinationFile;
   Process pc =
    Runtime.getRuntime().exec(commands);
   pc.waitFor();
  } catch (IOException e) {
  System.err.println(e);
  } catch (InterruptedException e) {
  System.out.println("");
  }
```

```
   }
}

// Server side thread
class Receiver extends Ddas implements Runnable {
   private Rendezvous r = null;
   public Receiver(Rendezvous r) {
     this.r = r;
     new Thread(this).start();
   }
  // running a thread for receiving integer data
   public void run() {
     setmsgReceived((String)r.serverGetRequest());
     r.serverMakeReply("received");
     r.close();
     setLock(1);
   }
}

// Client side thread
class Sender extends Ddas implements Runnable {
   private Rendezvous r = null;

   public Sender(Rendezvous r){
     this.r = r;

     new Thread(this).start();
   }

   // running a thread for transferring data
   public void run() {
     int breakTime = 1000;
     pause(breakTime);

     try{
        (String)r.clientMakeRequestAwaitReply (getmsgSend());
     } catch (MessagePassingException e) {
       System.err.println(e);
       r.close();
       System.exit(1);
     }
     pause(breakTime);
     r.close();
     setLock(0);
   }
}
```

```
package DM;
import java.io.*;
import java.util.*;

public class Decomposer implements ExecType{
   private String[] prog;
   private String[] type;
   private String[][] input;
   private String[][] output;
   private int count=0;
   private int dataCount=0;
   private String[] data;

   public Decomposer(int i, String[] program, String indata)throws Exception{
      prog=new String[i];
      type=new String[i];
      getData(indata);
      input=new String[i][dataCount];
      output=new String[i][dataCount];
      StringTokenizer str;
      count =i;

      for (int j=0; j<count; j++){
         str = new StringTokenizer(program[j], "_");
         prog[j]=str.nextToken();
         type[j]="_"+str.nextToken();

         if (j==0){
            for (int n=0; n<dataCount; n++){
               input[j][n]=data[n];
         }
         else{
            for (int n=0; n<dataCount; n++){
               input[j][n]="out"+(j-1)+n;
            }
         }
         for (int n=0; n<dataCount; n++){
            output[j][n]="out"+j+n;
         }
      }
   }

   private void getData(String indata) throws Exception{
      BufferedReader stdin = new BufferedReader(new FileReader(new File(indata)));
      StringTokenizer str=new StringTokenizer(stdin.readLine());
      dataCount=str.countTokens();
      data=new String[dataCount];
      for (int m=0; m<dataCount; m++){
         data[m]=str.nextToken();

      }
   }

   public void spawn(int index, int d) throws Exception{
      spawn(type[index],prog[index],input[index][d],output[index][d]);
   }

   private void spawn(String type, String prog, String input, String output)
     throws Exception {
     String command=null;
     try {
        if (type.equals(JAVACODE))
           command="Execj"+" "+prog+" "+input+" "+output;
        else if (type.equals(OTHER))
           command="Execc"+" "+prog+" "+input+" "+output;
        Process pc= Runtime.getRuntime().exec(command);
        pc.waitFor();
     } catch (IOException e) {
        System.err.println(e);
     } catch (InterruptedException e) {
        System.out.println("");
     }
   }

   public String getProg(int index){
      return prog[index];
   }

   public String getType(int index) {
      return type[index];
   }
   public String getInput(int index, int d){
      return input[index][d];
   }

   public String getOutput(int index,int d){
      return output[index][d];
   }

   public int getProgCount(){
      return count;
   }
```

```
  public int getDataCount(){
    return dataCount;
  }
}

public interface ExecType {
  final String JAVACODE="_j";
  final String OTHER="_c";
  final int FLAGLENGTH=2;
}
```

VITA γ

Zhilan Liu

Candidate for the Degree of

Master of Science

Thesis: A DISTRIBUTED SYSTEM FOR FUNCTIONAL COMPUTING

Major Field: Computer Science

Biographical:

Education: Graduated from No. 1 Middle school, Guangze, Fujian, P.R. China in 1988; received Bachelor of Engineering degree in Industrial Management Engineering from Northwest Fangzhi University, Xi'an, P.R. China in July 1992; Completed the requirements for the Master of Science degree with a major in Computer Science at Oklahoma State University in May 2001.

Experience: Graduate teaching assistant/computer lab team leader/computer assistant of Oklahoma State University, from August 1998 to December 2000. Employed as a statistician/DBA/MIS engineer by Nanfang Co., Ltd., Fujian, P.R. China, from 1992 to 1997.