Automatic Tools for Extending Network Computation Capabilities of HLL

By
Abdulla M. Abu-Ayyash

Supervisor
Dr. Riad Jabri

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

Faculty of Graduate Studies
University of Jordan

Aug 1999

This thesis was successfully defended and approved on Aug 16th. 1999

Examination Committee	**Signature**
Dr. Riad Jabri, Chairman Assoc. Prof. Of Compiler Design	[illegible]
Dr. Ahmed Al-Jaber, member Assoc. Prof. of Algorithms	[illegible]
Dr. Ahmad Sharieh, member Ass. Prof. Of Parallel Processing	[illegible]
Dr. Munib Qutaishat, member Ass. Prof. Of Data Base Management Systems	[illegible]
Dr. Abdulraouf Yousef Al-Hallaq Ass. Prof. of Computer Networks.	[illegible]

Acknowledgements

I would like to express my special warm thanks to my supervisor Dr. Riad Jabri, for his patience and guiding hand. Thanks also to the discussion committee Dr. Munib Qutaishat, Dr. Ahmed Al-Jaber, Dr. Ahmad al-sharaiah and Dr. Abdulraouf Al-Hallaq for their enrichment comments.

A very special warm thanks to Husam Saadeh, Mohanad Al-Selawi and Einas Al-Omari for their encouragement and continuous support. Thanks to my colleagues Amjad Hudeb, Bian Shawer and Ansar Majdalina for their spiritual support. To all I would say, "Brothers and Sisters"

Thanks also to my mother, father, brother and sisters for providing me with the encouragement, support and time when I needed it.

List of contents

List of Tables

List of Figures

Appendices

Abstract

Automatic Tools for Extending Network Computation Capabilities of High Level Languages

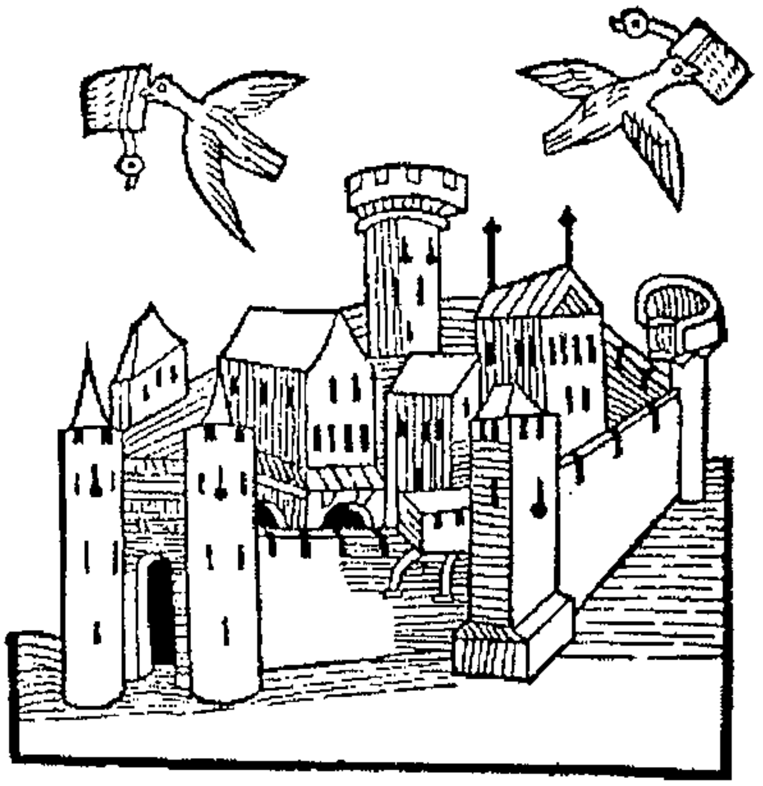
By
Abdulla M. Abu-Ayyash

Supervisor
Dr. Riad Jabri

Network computations have been under study for several years. Now, it is one of the most demanding subjects for information technology developers and users. The main idea behind network computation (NC) is how to do computation over the network The Internet hype is one example that makes the demand high. Today multiple methods are being used to make use of network computation, non of them tackles the legacy high-level language programs. In this thesis a study about network computation models are investigated. A new classification is proposed and a new network computation model for legacy high-level language programs is proposed named "Implantation".

In this study, different models for network computations is investigated and classified, using its properties, applications and programming languages. Different classification for computation models is studied, and showed that such classifications are weak to classify the network computation models. A new classification is introduced and named CDP-Classifications.

We implement the new model for NC (Implantation) using the virtual machine of imperative HLL. Such model can be useful for deploying legacy programs (written in imperative HLL) to make use of network computations. These computations can be used to utilize the available network resources. In this model, HLL programs would be executed using multiple nodes within a network.



Pigeon Post, Woodcut from 1481.
Coll. Bibl. de Genève, Fabre 1963
"The Early History of Data Networks Napoleon's Internet", 1994

Chapter 1

Introduction

1.1 History

At the first days of computers and around 1940, computations were mainly designed with the centralized computer in mind. Programming languages design was also affected with such a paradigm. Around that period the development of programming languages motivation was how to use the von Neumann architecture (Georg, 1987) to solve numerical problems. FORTRAN founded in 1956 (Alice, 1993) was the first language to implement such a paradigm.

HLL programs were designed with two assumptions in mind: code will be executed on the same machine during all execution time, and data is always available to the program during execution. As new architectures (distributed systems, parallel computers, data flow computer) (Barry, 1996)(Andrw, 1995) and computer networks systems have been introduced; these two assumptions are not applicable. Different parts of the program can be executed on different machines and data can be accessed on demand. This set new design goals for HLLs. New programming languages have appeared to fulfill these goals and needs of architectures and the computer networking. Examples of HLLs are Java (Jams, 1996) for network programming languages, CHARM (Kale, 1994) for parallel programming languages, Maisie (Rajive, 1995) for simulation languages and Telescript (Peter, 1996) for mobile programming languages (Tommy, 1997).

Furthermore, to fulfill the need for simplicity and portability, some standards have emerged based on remote procedure call RPC and distributed object technology. The RPC was introduced by Birrell and Nelson (1984). In their model procedure resides on remote servers are executed as if they resides locally. Combining object-oriented paradigm with RPC architecture named distributed object technology serve as a solution to solve the problem of portability and interoperability of applications over the network. Two distributed object models have emerged as open standard specifications: Common Object Request Broker Architecture CORBA and Distributed Component Object Model DCOM (Chung, 1997).

As the networks are widely used, they became the backbone for modern programming languages and applications. This led to the introduction of network computation models to overcome the limited (absence) networking capabilities of HLLs.

Most computation models suggest new programming languages rather than extending network computation capabilities of the existing HLLs. The main objective of this thesis is to address this issue by introducing and implementing automated tools to facilitate the use of existing HLLs in network computation and thus adapting their respective legacy systems to the network environment. These tools are based on new network computation model. Within the framework of this model an execution environment is introduced that permits implantation of the execution units EU from HLL into network computation nodes (NCN).

The Implantation concept consists of decomposing the HLL into execution units (EU). These units are migrated from their native (home) nodes to different nodes of the network for their execution. The native nodes module are called NC*c* while the destination node modules are called NC*s*, *see fig (1)*

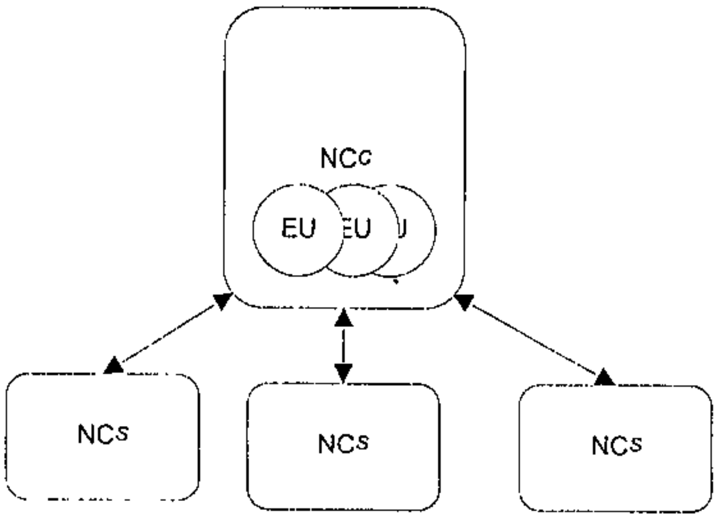


Fig (1): Abstract Implantation model of HLLs

1.2 Thesis Structure

This thesis is divided into the following chapters: chapter 2 is a background with available network computation models, classifications, properties, applications and available programming languages. Chapter 3 discuss a new classification of computation models. Chapter 4 describes the proposed network computation model for HLL (i.e., Implantation Model). Chapter 5 describes the implementation for a simplified two NCNs. Chapter 6 is the conclusion and suggested future work.

1.3 Contribution

This thesis contribute to the current network computation area specially in the following:

- Classification of computation models: two classifications regarding computational models were studied and shown that these models can not classify all known models of computation. A suggested more comprehensive classification be introduced.
- Network computation model for high level languages: we proposed, implement and test a new model for network computation to be used with legacy HLL programs. The model is named "Implantation".

Chapter 2

Background

2.1 Introduction:

This chapter discusses HLLs classical computation model, and available network computation models with examples for each.

2.2 HLLs and classical computation model

In this section, we will study the evolution of HLL, the design goals, classifications, and new challenges for modern computation.

2.2.1 Introduction:

During forty years of programming language development, multiple language designs were made to fulfill different aspects of problem solving. The number is increasing rapidly, for example during the period from 1950-1960 only four procedural languages was known, 1960-1970 around ten languages (Alice, 1993). At that time, almost all workers in the computer field knew all the languages available.

Today, there are hundreds of programming languages that are available, for sure no one can master them all. There are still new ones emerge to fulfill new design and architecture.

2.2.2 *HLL classification and design goals:*

Classifications of HLLs are always misleading. Languages are more alike than different, since the purpose of all languages is to communicate models from human to machine. In addition, category names are used loosely, and the same programming languages belong to more than one classification (Alice, 1993). Never the less, all HLLs have common design goals.

During forty years of language development, many features appeared to be a key design goals for programming languages, such as efficiency, portability, readability, modeling ability, simplicity, semantic clarity and safety (Alice, 1993). The meaning of each design goal will be discussed shortly.

Efficiency is related to how easy or difficult to translate any feature within the language into efficient code, and how to improve or degrade the performance of programs.

Portability is related to whether any feature within the language can be easily implemented on different machines.

Readability is related to whether any feature within the language make the program more readable. In addition, whether the programmer is able to understand such features.

Modeling ability is related to whether all features within the language is able to express the meaning of the program, and how easy it is to model all problems precisely and easily?

Simplicity is related to whether the language design is as a hole simple, unified, and general, or it is full of special-purpose features.

Lastly, semantic clarity is related to whether every legal program and expression have one defined, unambiguous meaning.

All the above and more design goals have been taken into account in designing classical HLLs. However, HLLs will need more design goals and challenges that will be discussed shortly.

2.2.3 New Challenges for HLL:

The design of HLL faces multiple challenges for new application area, in this section such challenges are discussed.

2.2.3.1 *New architectures*

Despite the fact that Flynn outlined in 1966 (George, 1987) four classes for the organization of high-speed computers, only this decade witness an accelerated development of programming languages for such architectures. Thus, programming language design for such architectures were developing slowly, that is because the cost of such machines is high and it is not widely available. As the wide spreading of networks, attempts was made so that the function of all machines within one network looks like one parallel virtual machine (PVM architecture) (Al Geist, 1994). Multiple languages was designed for that purpose like Orca (Henri, 1994).

2.2.3.2 *Standards and Portability:*

Although all programming language implementers claim standardization, different implementations do exist. For example, MS-Pascal, Turbo-Pascal and IBM-Pascal are different implementation for Pascal programming language but none of them implements standard Pascal. Each has its own features.

Due to the very high growing rate for use of networks, standards have become even more important feature for every tool that has to be used over the network, including programming languages and communication protocols. Such importance realization was the creation of Java programming languages and the adoption of TCP/IP (Fred, 1996) network protocol for Internet. Another method used for standardization of distribution of objects reusability was the adoption of CORBA and DCOM standards.

2.2.3.3 Network Models:

Despite that, all exiting network models have approximately the same objectives (connecting computers together) they are dislike. Such models differ on the roll of each node within the network, code and data residency, computation location and level of transparency for code and data migration. When network was first introduced, data was the first element to migrate from one environment to the other, so data transfer mechanism must be maintained. Now both data and code can migrate from one node to the other. And in a newly designed networks, even the network binders like routers and switches can be programmed using some programming languages like PLAN (Michael, 1998), such designs creates two types of networks, named active and passive networks.

Programming languages must have new features to support such the need to create a connection with other nodes over the network, and to send and receive both code and data.

2.2.3.4 New Applications:

Internet can be classified as the most important information explosion method in this decade, and for its fast web spread. New applications appeared, such as e-commerce, e-business, data mining, search engines and Intranet applications, … etc.

New programming features for database operations had to be added, like starting a client transaction and searching a database server.

2.2.3.5 Security:

Despite that, safety for long time has been one of the main design goals for HLLs. Security has long time been dropped of such design goals. That is because programs and data are supposed to be on one physically secure machine.

Today the story has changed, and security model has to be added to the programming language design goals, security constructs must be added specially for programming languages designed for network computation. Such features like data verification and user authentication.

Java, for example, added some security features that will prevent the program from affecting the execution environment or the internal state for the machine that is running on. This is accomplished by the introduction of sandbox operation model and byte code verification (Robert, 1998)(Jonathan, 1999). PLAN used authentication and authorization (Mechael, 1999).

Some new execution environments that emulate parallel computer over a network needs security measurements. While parallel programming languages in general do not need security support since the program will run on one physical machine.

2.3 Network Computation

Network computation is characterized by how to make computation over the network. In this section, some attempts for classifications are discussed to show that such classification lacks new classification. We will also discuss network computation languages that are available today and some execution environments for network computation with middle-tiers for network computations.

2.3.1 Network computation classifications:

Several attempts for classification of computation models was down by many researchers. Each attempt was modeled for specific type of application (Kristian, 1996)(Antonio et al, 1996).

Kristian Paul classification (1996): Kristian classifies computation models to be either centralized or distributed. Distributed model is divided into three classes workstation model, Xterminal-Server model and processor pool model.

Antonio et al classification (1996): proposed A new classification for modeling mobile code paradigms which he named *Client / Server (C/S)*, *Remote Evaluation (REV)*, *Code on Demand* (COD) and *Mobile Agent (MA)*. He also classified the mobile languages into two categories: strong mobile languages or weak mobile languages.

The strong mobile languages allow execution unites (EU) to be move with its internal state to a different site, like Telescript. The weak mobile languages allow EU to be bounded dynamically to code coming from different sites, like Java.

2.3.2 Programming Languages for network computation

In this section, a look on some network computation programming languages is conducted, describing the design goal and main features.

2.3.2.1 Java :

Java (James, 1996)(Sun, 1995) is a general-purpose, concurrent (multi-threaded), class-based, object-oriented language. It is designed to be simple enough that many programmers can achieve fluency in the language design. Goals of Java are to provide portability and robustness.

The Java is related to C and C++ but is organized rather differently. With a number of aspects of C and C++ omitted like multiple inheritance and pointer operations, few ideas from other languages were included like raising signals and catching exceptions. The Java is intended to be a production language not a research language. The design of Java has avoided including new and untested features. The Java is strongly typed language. Type checking is done on compile time and execution time. The Java does not include any unsafe constructs, such as array accesses without index checking, since such unsafe constructs would cause a program to behave in an unspecified way. Java is normally compiled to a byte code instruction set and binary format.

The Java was designed with network computation model in mind. It uses some API to establish network communication. It has remote method invocation (RMI) capabilities (Elliotte, 1997). The RMI offers some of the critical elements of a distributed object system, and it is analogous to CORBA and DCOM.

2.3.2.2 PLAN:

Programming language for active network (PLAN) (Pankaj,1999) is a new network programming language for programmable networks. These programs replace the packet headers (which can be viewed as very rudimentary programs) used in current networks (Pankaj, 1999).

The PLAN programs are lightweight and of restricted functionality. These limitations are mitigated by allowing the PLAN code to call node-resident service routines written in other, more powerful languages. This two-level architecture, in which the PLAN serves as a scripting or `glue' language for more general services, is the primary feature of this language.

The PLAN is based on the simply typed lambda calculus, and provides a restricted set of primitives and data types. The PLAN defines a special construct called a chunk used to describe the remote execution of the PLAN programs on other nodes.

Primitive operations on chunks are used to provide basic data transport in the network and to support layering of protocols. Remote execution can make debugging difficult, so PLAN provides strong static guarantees to the programmer, such as type safety.

A more novel property aimed at protecting network availability is a guarantee that PLAN programs use a bounded amount of network resources.

2.3.2.3 Telescript :

Telescript (Peter,1996) is an interpreted, dynamic, object-oriented programming language for writing mobile agents. A Telescript "program," or script, consists of a collection of classes. Classes are hierarchically organized by sub-classing. Classes have features that are their externally observable operations and attributes.

Features may be public or private. In contrast to the object models in some other languages, the Telescript's private features are visible in sub-classes as well as in the base class. Features, or entire classes, can be sealed so that they cannot be overridden in sub-classes or sub-classed, respectively. Telescript does not permit incompatible signatures (feature overloading) in sub-classes.

The Telescript language reference specifies a number of predefined classes. These must be supported in every implementation of a Telescript interpreter, or engine.

Objects are always instances of some class, and always inherit from the class Object, which is at the top of the object hierarchy. An object encapsulates its properties, specified by the class and directly accessible only to code "inside" the object (and not to sub-classes). Properties reference other objects. Agents and places are also objects, instances of various sub-classes of the predefined abstract class Process.

Process objects provide Telescipt's multi-tasking functionality. Processes are pre-emptively multi-tasked, and scheduled according to priority. Process classes have a live operation which is invoked directly by the engine, on a new thread of control, to animate new process objects.

All objects must be owned by at most one process. Objects that are not owned are subject to being garbage collected. A process "owns" itself. A process can transfer ownership of an object to another process, provided it owns the object and all objects referenced by that object's properties, and, recursively, all objects referenced by objects in that object's closure.

The Agent class, a sub-class of Process class, provides the sealed, private go operation. An agent can request the go operation on itself, providing a ticket argument. The ticket object describes the place where the agent is trying to go, with varying levels of specificity. If the engine can figure out where and how to route the agent, and the trip is ultimately successful, the agent "wakes up" executing its next line of code in the destination place.

Agent processes always execute in the context of one or more enclosing place processes. Places usually provide a service API for agents to interact with. Places can be nested within other places, and every engine has an engine place as its outermost place. As one of the last steps in processing an agent's go, the destination engine requests the entering operation on the destination place to give that place the opportunity to deny occupancy. If no place satisfying the ticket will admit the agent, the trip fails with a trip exception.

2.3.3 Execution Environment for network computations.

2.3.3.1 PVM :

Parallel virtual machine (PVM) *(*Al Geist, 1996*)* is a result of trying to use the computation power of all computers within a network framework. It was designed to permits a heterogeneous collection of Unix computers networked to be viewed by a user's program as a single parallel computer.

PVM is the mainstay of the Heterogeneous Network Computing research project, a collaborative venture between Oak Ridge National Laboratory, the University of Tennessee, Emory University, and Carnegie Mellon University (Al Geist, 1996).

The PVM system has evolved in the past several years into a viable technology. It is used for distributed and parallel processing in a variety of disciplines. PVM supports a straight forward but functionally complete message-passing model.

PVM is designed to link computing resources and provide users with a parallel platform for running their computer applications. Irrespective of the number of different computers, their use and where the computers are located. It is capable of harnessing the combined resources of typically heterogeneous networked computing platforms to deliver high levels of performance and functionality.

Programs written in C, C++ or FORTRAN are provided access to PVM using calls to PVM library routines for functions such as process initiation, message transmission and reception and synchronization.

Major features of PVM are portability, heterogeneous, scalable, dynamic configuration, fault tolerance, dynamic process groups, signals, multiple message buffers, tracing and customized.

2.3.3.2 MAP:

Mobile agent platform (MAP) (G. Adorni, 1993) is a platform for the development and the management of mobile agents that gives all the primitives needed for their creation, execution, communication and migration.

A node belonging to the platform MAP consists of an object called server that contains, all the entities needed for the operation of the platform itself. In the host there can be more than one server.

Server is the main object of a MAP server, in which the entities Daemon can Context and local agents are instanced. Its activation enables the node to accept and have agents coming from the network run.

Daemon is the entity of the MAP that listens on a certain port, waiting for agents coming from other nodes and for messages to be liveried to local agents.

Context is one of the basic objects in a MAP server. In fact, it knows all the agents present on the server.

Network Class Loader is used for enabling the agents to run on a specific MAP server, even when their class is not present there.

Code Server is an internal entity of the Context, dynamically created. In fact, the Context of a platform instances a new object Code Server each time it is requested a class by a network Class Loader.

The most important feature of the MAP platform is the possibility to make an agent migrate or to send messages through the network even to servers whose classes to which such objects refer are not present. To do this, each time a specific object reaches a new server, the corresponding Daemon load a new Instancer object with a Network Class Loader that deals with the loading of such classes. If they are not available locally, the Network Class Loader interrogates the Code Server of the some remote sites saved on an appropriate vector, searching for the classes required. If it finds them, they are loaded from the remote site and saved in the local table of classes.

MAP environment implementation was written fully in Java, and that is because Java supports object-oriented, multithreaded language and for its portability feature.

2.3.4 *Middle-tiers for network computations*

To simplify network programming, the RPC was introduced by Birrell and Nelson (1984). To realize component-based software architecture, two distributed object models have emerged as standards namely, CORBA and DCOM (Chung, 1997). In the following three sub sections, we will present a short description for each.

2.3.4.1 *RPC:*

Remote procedure call (RPC) (Tanenbaum, 1995) is an older model than both CORBA and DCOM. While CORBA and DCOM create objects remotely, the RPC is a method that allows programs to call procedures located on other machines.

RPC achieves transparency by using both client and a server stubs. Client stub function is to pack parameters into a message and asks the kernel to send it to the server stub. At the server stub, the stub unpacks the parameter and calls the server procedure. After the end of the procedure, the server stub packs the return results and sends the message to the kernel to deliver it to the client stub. At the client stub, the result is unpacked and the program continues to execute.

This model was efficient for none object-oriented methodology. Today, object-oriented paradigm is seen as the key to a successful semantic clear computation. For that, RPC is no more the key to a successful computation. New models has been proposed called CORBA and DCOM, which will be discussed in the following two sub-sections.

2.3.4.2 CORBA:

Component Object Request Broker Architecture (CORBA) (Chung, 1997) is a distributed object framework proposed by a consortium of many companies called Object Management Group (OMG).

The core of CORBA architecture is the Object Request Broker (ORB) that acts as the object bus over which objects transparently interact with other objects located locally or remotely.

A CORBA object is represented to the outside world by an interface and a set of methods. A particular instance of an object is identified by an object reference. The client of a CORBA object acquires its object reference and uses it as a handle to make method calls, as if the object is located in the client's address space.

The CORBA framework for distributing objects consists of the following elements: ORB, IDL, DII and IIOP.

The IDL is an Interface Definition Language for defining static interface, DII is Dynamic Invocation Interface which let clients access objects. And IIOP is Internet Inter-ORB Protocol. A binary protocol for communication between ORBs.

The CORBA is a promising technology that is seen as the future computation model that enables application to communicate and share objects.

Despite that, CORBA is a more generic specification for distributed object technology. Microsoft implemented a different model named DCOM that will be discussed in the following section.

2.3.4.3 DCOM:

Distributed Component Object Model (DCOM) (Chung 1997) is a distributed extension to Microsoft component object model (COM) that build an object remote procedure call (ORPC) layer on top of RPC to support remote objects. A COM server can create object instances of multiple object classes. A COM object can support multiple interfaces each representing a different view or behavior of the object.

An interface consists of a set of functionally related methods. A COM client interacts with COM objects by acquiring a pointer to one of the object's interfaces and invoking methods through that pointer, as if the object follows a standard memory layout.

2.4 Summary:

The HLL designed goals has changed during the past forty years, new challenges emerged such as new architectures and configurations, portability and the need for standard, new emerged network models, new applications and security.

Network computation model relies on how to do computation over the network. Two classifications were discussed.

Some programming languages for network computation were also studied, like Java, PLAN and Telescript. Two-execution environments were discussed PVM and MAP. Some middle tier paradigm has been studied like RPC, CORBA and DCOM.

2.5 conclusion

Multiple methods emerged to address the new challenges for HLL, such as designing new languages, introducing new execution environments with library routines extension and using middle-tier paradigms. None of them addresses the problem of using classical legacy HLL programs with the network environment. We suggest extending the HLL capabilities for network computation by designing new execution environment. It should be transparent to both the user and the HLL program. That is to say, the HLL program should execute correctly in the new environment without changing its semantic.

In an attempt to classify all network computation methodologies studied, we found some classifications that lack's some models. Such classification will be discussed in the following chapter.

Chapter 3

New NCM Classification (CDP-Classification)

3.1 Introduction

Several attempts for classification of computation models using existing paradigms has been conducted by may researchers. In this chapter, two attempts by Kristian and Antonio et al will be discussed. The discussion will focus on limitation of each to fulfill the complete classification, and conclude with a new classification.

3.2 Kristian classification

Kristian (1996) in his thesis classifies models of computation to be either centralized system model or distributed system model. The distributed system model can be workstation model, Xterminal-Server Model and Processor pool model.

The centralized system model consists of a set of terminals connected to a single central computer. The central computer is typically a timesharing system which switches frequently between processes to provide a share of the system resources.

The distributed system model is directed by two technological advances: the evolution of cheap powerful CPUs and the advent of high speed networks.

The following is Kristian classification for distributed system computing:

1- Work station model: The workstation model is a group of personal workstations, connected by a high-speed network

2- Xterminal-Server-Model: Xterminal-Server Model is a hybrid of the workstation and timesharing models.

3- Processor Pool: Processor pool model is another form of distributed system developed to exploit the availability of small cheap microprocessors.

Kristian classification is based on the topology for distributed computing. It is directed by the current computation technology. Such model is weak to describe the network computation models, since it is directed by the distributed system computing point of view.

3.3 Antonio et al. classification:

A more general classification using mobile agent paradigm was discussed by Antonio et al. (1996). Their model classifies mobile code paradigms based on interactions patterns that define the coordination and relocation of components needed to perform a service. Their classification is based on the following abstractions: components that can be either resource components or computational components, Interactions and Sites.

Resource component embodies architecture elements representing passive data or physical devices, like file, a network device driver, or a printer driver. A particular kind of resource is represented by code components that contain the know-how.

Computational components embody a flow of control, an example is a process, or a thread. They are characterized by a state of their execution.

Interactions are events that involve two or more components. The sites are execution environments; they host components and provide support for the execution of computational components.

In their classification, they suggested that there are three main design paradigms that extend the well-known client-server paradigm; remote evaluation (REV), code on demand (COD) and mobile agent (MA). They distinguish the design paradigms according to the location of the different components before and after the execution of the service. See Table (1).

In Client/Server paradigm, a computational component B (the server) offers a set of services that is placed on site Sb. Resources and know-how are hosted bye site Sb. The client component A located on Site Sa, requests the execution of a service with an interaction with the server component B. As a response, B performs the service requested by execution the corresponding know-how and accessing the involved resource. If result is needed it will be sent back. For short client's viewpoint that the server owns all necessary data and knowledge.

In Remote Evaluation paradigm, a component A has the know-how necessary to perform the service but it lacks the required resources. Which happed to be located at a remote site Sb. Consequently, A sends the service-Know-how to a computational component B (Called executor) locate at the remote site that, in turn, executes the code using the resources available there.

In Code on Demand paradigm, component A is already able to access the resources it needs, which are co-located with it within Sa. However, no information about how to process such resources is available at S1. Thus, A interacts with a component B contained in Sb by requesting the service know-how, which is in Sb as well. A second interaction takes place when B delivers the know-how to A, which can subsequently execute it.

In Mobile Agent paradigm, the service know-how is owned by A, which is initially hosted by Sa, but some of the required resources are located on Sb. Hence. A migrates to Sb carrying the know-how and possibly some intermediate results with itself. After it has moved to Sb, A completes the service using the resources available there.The following table summarizes their classification:

Table (1): Antonio et al. Classification for mobile code paradigm

Paradigm	Before		After	
	Sa	Sb	Sa	Sb
Client/Server (C/S)	A	Know-How Resource B	A	Know-How Resource B
Remote Evaluation (REV)	Know-How A	Resource B	A	Know-How Resource B
Code on Demand (COD)	Resource A	Know-How B	Resource Know-How A	B
Mobile Agent (MA)	Know-How A	Some Resource	-	Know-How Resource A

As we can see, their classification is based on mainly two elements computational component and resource components, which leads to four classifications.

3.4 Suggested computation model classification (CDP-Classification)

Antonio et al classification seams closer to the model of network computation than Kristian, but it can not fully reflect all computational models. That is because resources never move, in real life, data resource can migrate, especially in database applications. Therefore, we need a new more general classification.

After a deep study and analysis of computation components we decided to divide it to its basic elements: code, data and processing. Then we suggested that any element within a network could be founded locally or remotely. We also assumed that all elements of resources must be available to the computation before it starts, that summed to eight computational model classes. As shown in table (2).

1-Centralized Computational class (CC) is when all computational elements are located locally. No need to use network to migrate resources. It is the oldest computational model of HLL.

2-Data Fetch on Demand class (DOD) is when all computational elements reside locally except data that is fetched from remote site on demand. Database inquiry is an example for such model.

3- Code on Demand class (COD) is when all computational elements reside locally except the code that is fetched on demand from the remote site. HTML and Java scripts are good examples for that.

4-Object on Demand class (OOD) is when objects (code and data) reside on remote machine and execution will be carried locally, Java applets with Java database connectivity JDBC.

5-Remote Object Execution class (ROE) is when objects are transmitted to be remotely executed, mobile agent in a way or another can be seen as an example, we will discuss mobile agents in more details shortly.

6-Remote Code Execution class (RCE) is when the code will be transferred to be executed on remote machine, our model of implantation is one example.

7-Remote Data Processing class (RDP) is when data is sent to remote machine to be processed there. Database applications that resides on servers that collect data to be processed later is a good example for that.

8-Client/Server class (C/S) is when the remote machine has all the computation elements and provides services to clients; RPC is one good example for that.

Table (2): CDP-Classification

Code	**Data**	**Processing**	**Model Name**
Local	Local	Local	Centralized Computation model (CC)
Local	Remote	Local	Data Fetch on Demand (DOD)
Remote	Local	Local	Code on Demand (COD)
Remote	Remote	Local	Object on Demand (OOD)
Local	Local	Remote	Remote Object Execution. (ROE)
Local	Remote	Remote	Remote Code Execution (RCE)
Remote	Local	Remote	Remote Data Processing (RDP)
Remote	Remote	Remote	Client/Server (C/S)

3.5 Antonio et al. Mobile code paradigm and our classification

Despite that our CM classification appears not to classify mobile agent paradigm, it can easily classify such paradigm. As Antonio et al (1996) states mobile code paradigm. The computation as a hole initially starts locally, on a certain point, all the computation with its internal state and temporary results is transferred to remote node to continue execution. So any migration of code from node to node can be seen as a mobile code paradigm. In addition, from our classification we see that their classification for their paradigms can be found in our classifications; C/S, RCE, COD and ROE respectively.

3.6 Conclusion

Kristian and Antonio et al. Classification is not suitable to classify network computation models. The suggested model for network computation we think is complete. the complete eight classification is named CDP-Classification while the last seven elements is named network computation model NCM-Classification. Such classification as of our opinion can classify all existing computation models, including the new emerging mobile code paradigm.

Chapter 4

Implantation Model

The classical computation model of HLL is based on the following:

- The semantics of HLL: the semantic is defined in terms of operations and data structure of a virtual machine (VM). The compilation process of HLL is then reduced to mapping of a program written in these languages into equivalent ones in the machine language of the VM.
- VM of compiled HLL: it is the hardware interpreter of actual machine and hence its machine language is the actual machine language. A VM of interpreted language is a software-simulated machine and hence a machine language might have any form.

Both implementation models of HLL are platform dependent and require both data and code to reside on the same machine where the code is executed.

Actual machines are no more isolated. They are connected within networks. For this reason, code and data can reside on any node within the network and they can be moved while the program is executed. A research effort for language designers is directed towards assuming that the code and data can reside on any node within the network.

4.1 Known efforts to make use of NC models in HLLs

An attempt for extending HLL for network support was by adding some communication libraries for existing languages such as C/C++ with WinSock Library (Arthur, 1995). Although same syntax and semantic are preserved for HLL, ones have to learn how to use such libraries. It is difficult some times to express fully the model required, and some times, it is impossible to express such models. For example, you can not express mobile code in C with network routines. C is not a byte code compatible with different platforms.

Another attempt was by modifying existing HLL to reflect some specific models of computation like parallel programming (Russian, 1997). In this approach, not all constructs within the language are new to the user, it is more attractive, and it is easy to learn. A successful example is pC++ and CC++ (Doreen, 1993). Again, the hosting language semantic has to change to reflect the new model. Thus, programmers will see conflicts in having the same syntax but deferent semantics. More over such languages have some limitations for modeling capabilities. Therefore, the solution of such problem can be either by introduction of new programming languages or new transparent execution environment.

New programming languages like Java (James, 1996) and PLAN (Michael, 1998) have new syntax and semantics that directly reflect the network computation model. Such languages are easy to use by experience user. It can easily express the model that was designed for. However, a new language must be learned, which is not an attractive idea.

While new programming languages is the solution for new requirements, legacy HLL programs can not benefit from such languages. New transparent execution environment is seen as the solution. Network execution environment is the solution for extending legacy HLL programs to make use of network computation paradigm without changing the HLL semantic.

4.2 HLL and network computation

Exiting legacy systems and HLL has been for long in the market, they lack the capability of using networks during computation. We think that networks can give more power to such languages. Executing such programs within the network can utilize the network resources, such as processor power. It can increase the availability and use of such HLL. Our main objective is to enable classical legacy HLL programs to be adapted to networking environment during their execution.

We do not mean for sure to allow such programs to run in parallel on deferent machines. That is because such applications were not designed for such a paradigm, and for that it need a lot of automatic code analysis, before code fragmentation can be done. We mean that the HLL is implanted all over the network within a framework of network computation nodes (NCN).

So what is the Implantation model? Why it is proposed? and how HLL programs can make use of it? In the following section, the model will be discussed.

4.3 Implantation Model

The suggested model is based on basic idea of the so called Implantation, which means migrating the compiled code of HLL from the machine for which it is originally generated into another machine having the same execution environment of HLL. This migration process acts as planting the HLL on different network computation nodes. Permitting the programs written in HLL to be compiled and loaded in one machine (node), and then to be migrated as a whole or as execution units into another node(s).

Based on this idea, a suggested model, which will work in a network, consists of a set of network computation nodes. In each node, there are two types of network computation modules, named network computation server (NC*s*) and network computation client (NC*c*).

The function of the NC*c* is to execute the program and on a specific point (EU call). The EU will be transferred to any available NC*s*, remotely located on another NCN. The EU will be executed there and returning the results back then the program can continue execution.

4.3.1 Basic Concepts

Implantation environment consists of a set of nodes called network computation nodes. In each node, there are two types of network computation modules, named network computation server (NC*s*) and network computation client (NC*c*). The two modules are completely isolated, so the node can be doing some network computation for some other nodes and at the same time asking other nodes to do computation with.

In this section every element of the implantation model is discussed, network computation node, network computation server, network computation client and network computation registry.

4.3.2 Network computation node (NCN)

Network computation node is the core node for implantation model. It is the execution environment in which programs runs. It consists of two modules named network computation client (NC*c*) and network computation server (NC*s*). The set of network computation nodes within one location is called network computation network (NCNet). See fig (2).

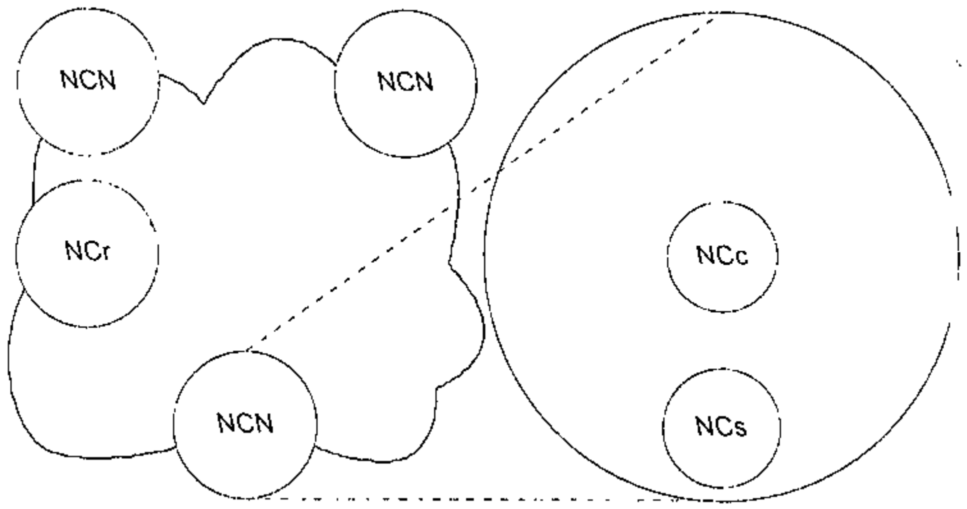


Fig (2): Network Computation Network (NCNet).

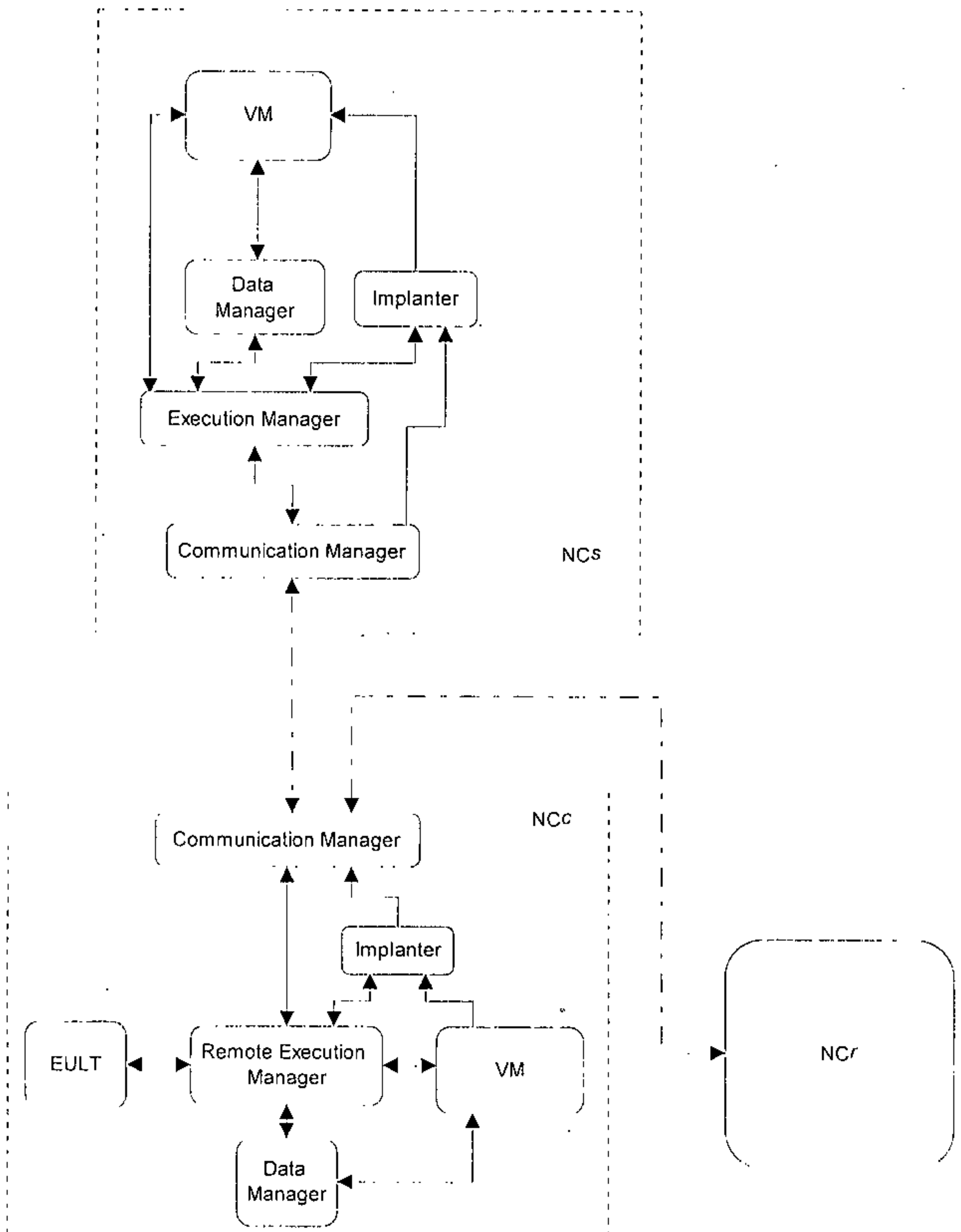
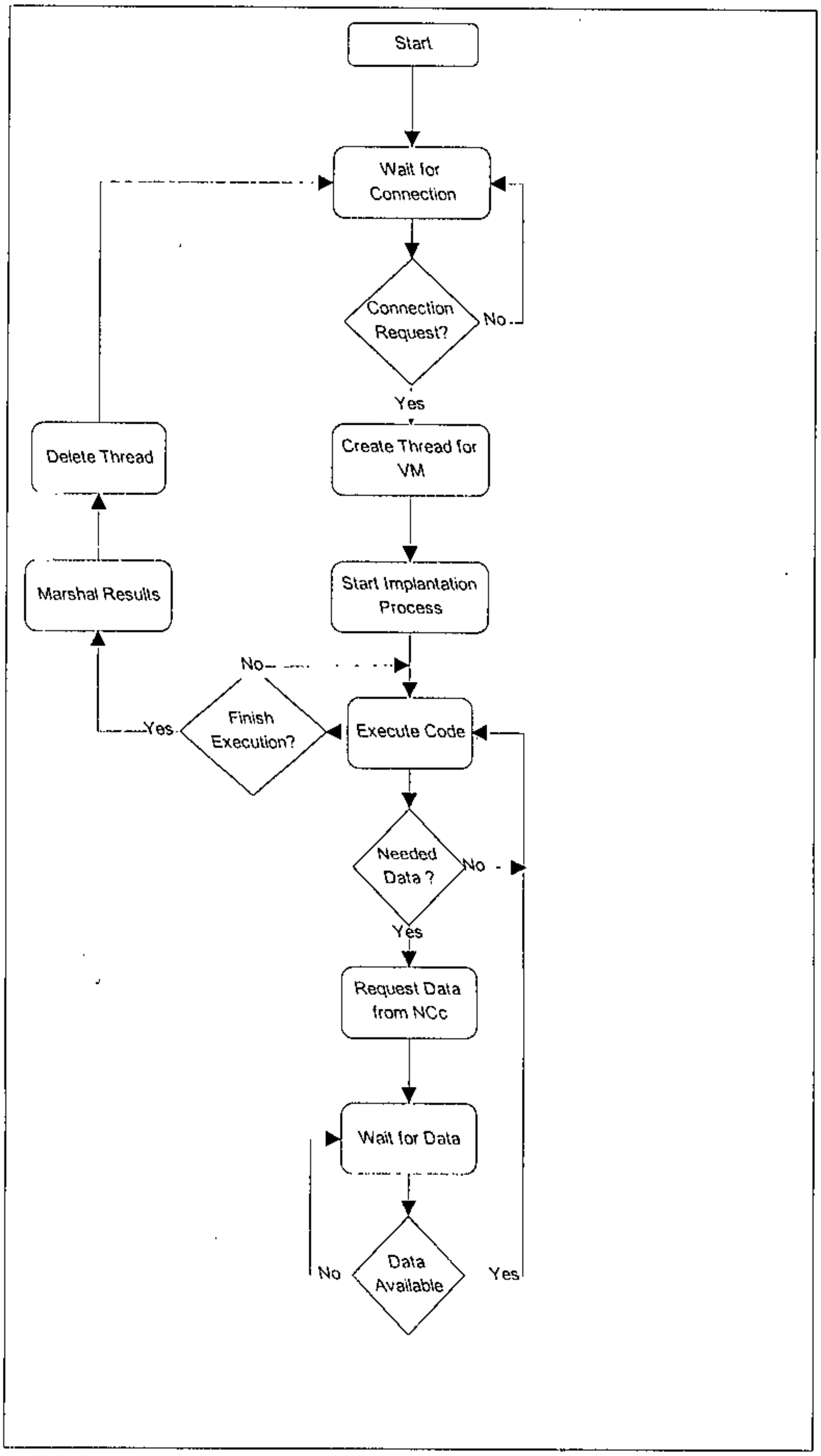


Fig (3): NC*s* , NC*c* components and NC*r* relationship.



Fig(4): NCs execution flowchart

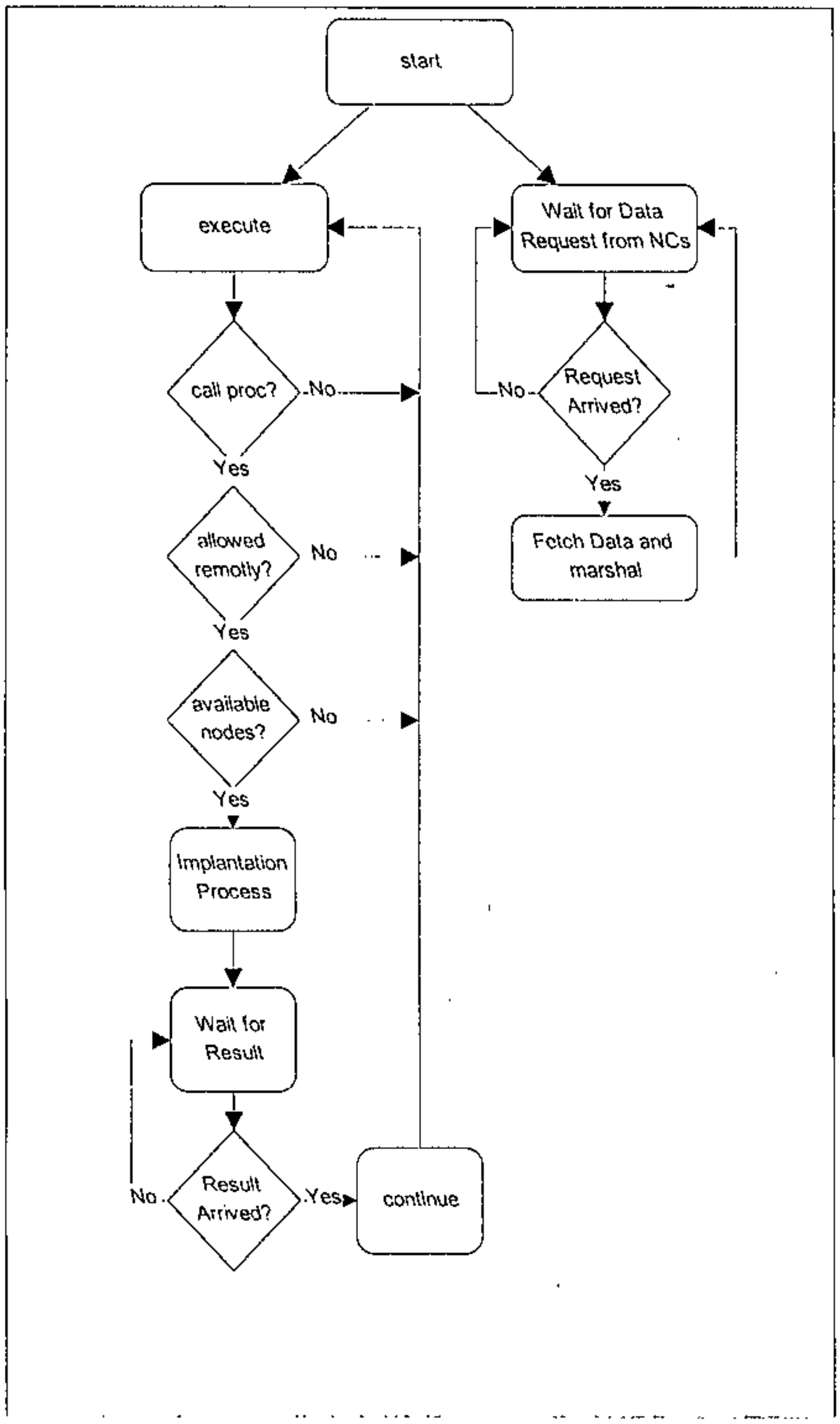


Fig (5): NC*c* execution flowchart

4.3.2.1 Network computation server (NCs)

Network computation server is the main execution environment for the implanted code. It consists of a virtual machine, server implanter, communication manager and data manager, see Fig 3. Where:

1) *Virtual Machine*: is the execution environment in which the implanted code of HLL will be executed. It communicates with the execution manager, the server implanter and the data manager.

2) *Server Implanter:* its function is to implant the code into the server virtual machine VM. After the implantation is finished it informs the execution manager.

3) *Communication manager*: is responsible for establishing the connections with the NC*c* and maintaining a communication channel with that node for message transfer during all execution time.

4) *Data manager* is responsible for fetching the data from the implanter node, when the VM asks for. It contains a result manager that will marshal the result to the implanter node (NC*c*)

5) *Execution* manager: is responsible for organization of all server components functions.

4.3.2.2 Network computation client (NCc).

Network computation client is the node responsible to initiate the implantation process. It consists of implanter, a virtual machine, communication manager, data manager and execution unit lookup table. See Fig 3.Where:

1) *Virtual Machine*: is the execution environment of the HLL local code.

2) *Implanter*: is the manager of implantation and responsible for transferring the code to the remote node.

3) *Communication manager*: is responsible for establishing the connections with other NCNs and maintaining a communication channel with that node for message transfer during all execution time. It uses the address that returns from the NCr. If a universal resource locator URL is used then it will be resolved for its Internet protocol address IP using distributed name server DNS.

4) *Data manager* is responsible for sending the data that,is requested by the NC*s*.

5) *Execution Unit Lookup Table (EULT):* A table within each NC*c* that is filled with information about which EUs are allowed to execute remotely. The table can be filled either explicitly or implicitly: explicit filling is controlled by user intervention, while implicit filling is a result of automatic code analysis.

6) *Execution* Manager: is the organizer between all other components.

4.3.3 Network computation registry (NCr)

Network computation registry is responsible for providing other NCNs with information about what NCNs are within the NCNet. It is a server which is selected within the NCNet to hold an entry for each created NC*s*, this registry is asked by each NC*c* for the availability of nodes to do network computation with. This is done to minimize allocation time for available NC*s*.

Each newly created node within the network must register its URL or IP address within the registry. To find the computation power for each NCN, it is suggested that the NCr will do a small computation with that node to find the execution time and communication time, these information is stored in a table. Based on this table nodes can be easily selected for its computation power.

4.3.4 Network execution environment of HLL and execution scenario

The model works as follows, see Figures 2,3,4 and 5. First, HLL program will be implanted (loaded) in NC*c*'*s* VM, then EU lookup table (EULT) is filled by the user with the allowed EU to be remotely executed, by default all EU are allowed to be remotely executed. Then execution is instructed by the remote execution manager within NC*c*.

The program will execute until it calls an EU, at that time execution unit in VM informs the remote execution manager that a call for an EU is encounter. The remote execution manager looks up into the EULT to see if the EU is allowed to be remotely executed. If not, the remote execution manager instruct the virtual machine to continue local execution. Otherwise, A call to network computation registry NC*r* is done to look for available network computation servers NC*s* to do computation with. If no nodes are available, then execution will continue locally, otherwise an available node address is sent back by the registry and the implantation process of the EU is started. Local execution will suspend.

After the end of implantation, the NCs starts executing the implanted code using its VM. If any data is needed, execution will suspend and a message is sent to NC*c* to send the desired data. After the arrival of data, execution is continued. At the end of execution of the implanted code, the NCs marshals back the result. When the result arrives to the local node (NCc) the execution is continued localy.

4.3.5 Fault Tolerance

Every time the NCc asks a node to do computation, it starts a time-out counter, if it expires before the NCs return the result then it starts another implantation with other node. If no nodes are available then local execution will continue.

Deadlock is not to happen regarding our model, and that is because at a certain time no two nodes are doing computation for the same HLL program. Resource race conditions may appear between different HLL programs. Time-out counter is used which also prevents deadlock.

The Implantation model has the feature that all needed resources are available locally and prior to computation. Other nodes are needed if they are available and have more power than local node.

4.4 Comparison between Implantation and RPC

If we assume that the implanted EU is on the server before a call was made then the calling process will look like RPC. That is why we compared the two paradigms. We will look at similarities and differences between the two models, its application, failure resistance, load balancing, platform dependency, called procedure implementation dependency and, language dependency. See table (3).

RPC is a language independent, that is to say, you can call a remote procedure written in another language, while Implantation will implant the procedure of the current program to remote node before execution. The code that is implanted is language dependent.

RPC and Implantation are platform independent. The Implantation model depends on virtual machine implementation. The RPC is a specification that co-exists on multiple platforms, in spite that RPC has to do some data conversion between platforms.

Implantation has the disadvantage of more message sizes. The length of messages is code dependent, whereas the RPC messages is for procedure call and results returned. The Implantation has a communication overhead over RPC.

The RPC employs static load balancing. That is to say, procedures are place in advance and prior to calling on specific remote nodes (servers) and hence the call is made even if the server is overloaded. The Implantation has the capability to implant the code to any available NCN within the NCNet at run time, and hence it can decide before implanting if the server is overloaded to chose another server.

The RPC has difficulty in its semantic in the presence of failures. That is when the client is unable to locate the server, the request to server is lost, the replay is lost, the server crash after receiving the request and the client crashes after sending the request. For a comprehensive study see (Tanenbaum, 1996). In most cases, the problem is that the computation is not available locally so computation will not continue until a requested server is recovered. This problem is not found in the Implantation, since the code of the procedures is locally available. If a requested server crashes, a time-out is encountered and an alternative one is selected, if no one is available, the computation can continue locally.

The RPC depends on the implementation version for the procedure on the host, while The Implantation is not dependent on any implementation over the network.

Finally, the domain of application for the RPC is modern network computation languages, while the implantation is designed for legacy systems and HLLs. Below table 3 presents a summary for comparison features of the RPC and the Implantation models

Table (3): Comparison between RPC and Implantation

Feature	RPC	Implantation
Language Independent	Yes	No
Platform Independent	Yes	Yes
Called Procedure implementation dependent	Yes	No
Communication overhead	No	Yes
Load Balancing	Static	Dynamic
Fault Tolerance	No, failure of Remote Node stops execution, if no other node has the implementation of the procedure.	Yes, failure of remote node does not stop execution. In addition, the only halt is when the local node halts.
Domain of application	Modern NC languages	Legacy HLLs (application)

4.5 Implementation considerations

When thinking in implementing Implantation, one must decide on multiple issues, such as the implementation language for the model and method of transferring data between nodes. Should the network computation nodes best be threaded? What is the best method of allocation and selecting the network computation node to do computation with? Is it a proper idea to do automatic code analysis for procedures that will be allowed to be remotely executed?

In selecting implementation language, we think it is better to select a language which has new features, like object oriented, network programming, portability, supports multi-threading, and can easily facilitate different computation models.

There are multiple methods of selecting and allocating of network computation nodes within the NCNet. All have one of the following design decisions; deterministic versus heuristic, centralized versus distributed, optimal versus sub-optimal, local versus global and sender-initiated versus receiver-initiated algorithms. For a comprehensive study see (Tanenbaum, 1996). In our model we think centralized is the best for small NCNet.

Multiple methods for automatic code analysis for data dependency can be used. Most of them rely on generating dependency graph for basic blocks. Such dependency can be classified as output dependency, true dependency and anti-dependency (Reinhard, 1995). Code analysis can be done in two ways: dynamic (execution profiler) and Static (code analysis). Multiple tools are available to assists in such decisions. Automatic code analysis is costly in term of time that is why we think it is not feasible to implement.

4.6 Conclusion

We believe that the Implantation can make use of new network computation power available today for HLLs. That is because it does not change the semantic of the HLL programs. It has a failure recovery mechanism that is better than RPC.

Some drawbacks are in the model, like size of messages between nodes compared to RPC, but most mobile code paradigms today have such a problem, Java applets for example. Also it is a language dependent while RPC is not. RPC is not fully platform independent (Tanenbaum, 1995)

4.7 Summary

In this chapter, a study of the classical computation model of HLL was conducted. Such model is based on the assumption that both data and code will reside on the same machine all over the execution time of the program.

Known efforts to make use of network computation models in HLLs is discussed. Three methods were shown; extending language library, modifying existing HLLs and designing new HLLs.

Implantation model, the idea, abstract model requirements environment, and execution scenario was discussed in details.

A comparison between Implantation and RPC the most related computational model within our classification is conducted. Moreover, we found that Implantation is better than RPC in some issues like recovery capabilities, resource utilization and on dynamic load balancing. Most of the drawbacks are related to language dependency and size of messages.

Chapter 5

Implementation

In our implementation, we did not implement all suggested features for the model, our main goal was test the idea of Implantation. A compiler for a simplified Pascal programming language was written. The output is an assembly language for a stack-oriented machine. Such assembly code is implanted into the network computation node for execution. A stand-alone interpreter and a debugger for program execution were also written.

We did also implement the network computation node server (NC*s*) but without the data manager, and assume that the EU will be self-contained, that was done to simplify implementation. The server is named virtual machine server (VMServer).

We also implemented the network computation node client (NC*c*) also without data manager, and EULT was assumed to return that each procedure is allowed to be remotely executed. The client is named remote execution manager (RE).

We did not implement the network computation registry (NC*r*) since its function is to minimize the selection time of node to do computation with, so we assumed that local network computation node knows where is the network computation node servers.

This chapter is divided into the following sections: section 5.1 discuss the implementation language, section 5.2 discuss the compiler and the interpreter, section 5.3 discuss network computation server, section 5.4 discuss network computation client.

5.1 Implementation language

In selecting implementation language, we think that it is best to select a language that has new features like object oriented, network programming, portable, supports multi-threading, and can easily facilitate different computation models.

Object Oriented feature was selected since it can make reusing of modules easy and modeling is semantic clear. Network programming feature was selected to hide network protocol complexity when doing communication, so that our implementation will be network independent. Portability was selected so that we can extend our model to the biggest number of computer architecture available in the market today. Finally, Multi-threading is selected for utilization of computer computation resources, especially when implementing the server and network computation registry (NC*r*)

In spite that several languages has such features, we selected Java since it is one of the widely used programming language today. In addition, it proved that our choice was correct.

5.2 Simple Pascal Compiler, Interpreter and debugger

Simple Pascal was designed with the most essential constructs, no pointers, no goto, no complex types such as sets, no string manipulation, no case statement, no for loop, only while and repeat. For a complete list of implemented construct, see appendix A.

Simple Pascal compiler, interpreter and debugger was written in C language. C was used for its efficiency especially for system programming, and its portability feature over multiple platforms.

The stack machine selected for its simplicity was UGAVAC machine. It is a small virtual machine which has stack of size 46KB, a memory of 46KB, three special purpose registers; program counter PC, stack pointer SP and frame pointer FP, the ALU performs operation on stack arguments. For implemented instructions for the machine see appendix B.

5.3 VMServer: Simplified Network Computation Server

The simplified server is implemented with multi-threaded virtual machine, so it can server multiple requests at the same time. When the thread is created it will handle all future communication with the network computation node that requested the computation.

Server implanter function is to de-capsulate the implanted code to separate the code and the result location within the code, the byte code will be loaded into the VM and the result location will be sent to result manager. After the end of execution the execution manager is inform, which in its turn informs the result manager to get the result from the memory of VM and send it to the RE.

The following modules where implemented:

1) *Server Implanter:* its function is to de-capsulate the code and result location and implant the code into the virtual machine VM the result location is assumed to be the first variable defined within the EU.

2) *Virtual Machine*: is the execution environment, in which the implanted code of HLL will be executed, it is a stack-oriented machine.

3) *Communication manager*: is responsible for establishing the connections with the RE node and maintaining a communication channel with that node for message transfer during all execution time.

5.3.1 Pseudo code for the VMServer

```
Do forever {
 For each incoming connection request do
     Create VMs. Thread that do {
        Create connection with NCc.
        Wait for NCc. to implant code.
        Run the implanted code.
        Return the result.
        Close Connection with NCc.
        Stop tread.
     }
}
```

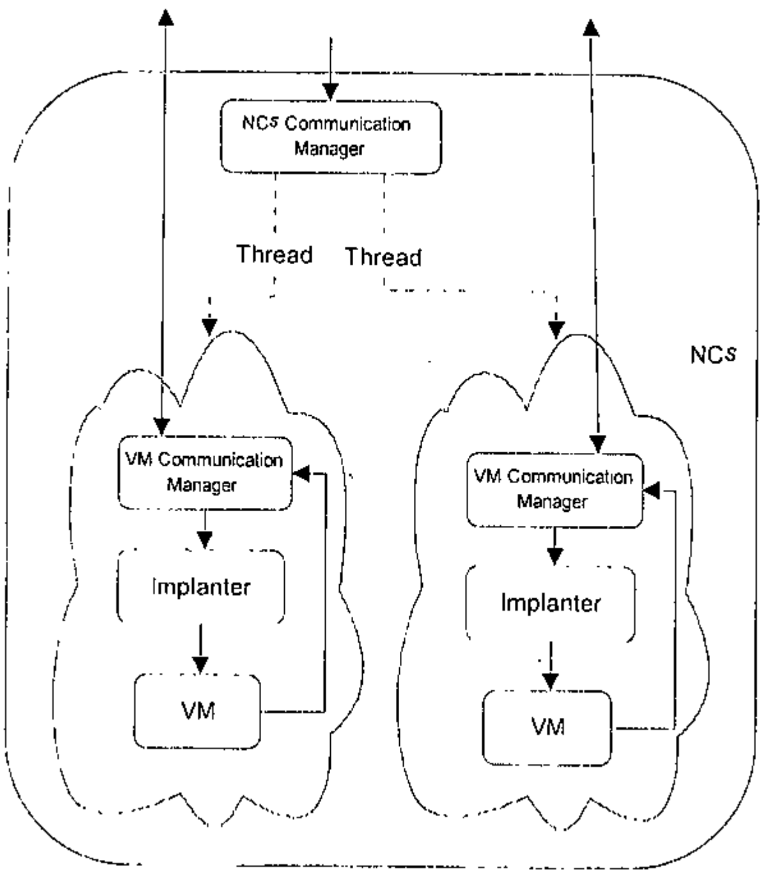


Fig (6): VMServer with threading.

After the connection request from the RE manager node, the VMServer creates a thread for VM and establish a channel between the two. After the communication channel is established the VMServer communication manager is no more responsible for message transfer between the RE node and VMServer, communication is managed by the VM communication manager.

5.4 RE manager: Simplified Network Computation Client (NCc)

RE is responsible to initiating the implantation process. It consists of implanter, a virtual machine, communication manager; we assumed that the full program is an EU that will be executed so implantation will be for the all the program, each element will be discussed below:

1) *Virtual Machine*: is the execution environment in which the implanted code will be executed, It is a stack oriented machine.

2) *Implanter*: is the manager of implantation and responsible for transferring the encapsulated message of code and location of needed result to the remote NCs.

3) *Communication manager*: is responsible for establishing the connections with other NCNs and maintaining a communication channel with that node for message transfer during all execution time.

5.4.1 Pseudo code for RE manager.

```
While not reached end of execution (Implanted code) do {
   Fetch instruction
  If instruction is a call to execution unit Do
          Make connection with NCs .
          Ask server to switch for implant mode.
          Implant EU.
          Is EU implanted?.
          Run EU.
          Wait for result.
          Close connection with NCs.
  Execute instruction.
}
```

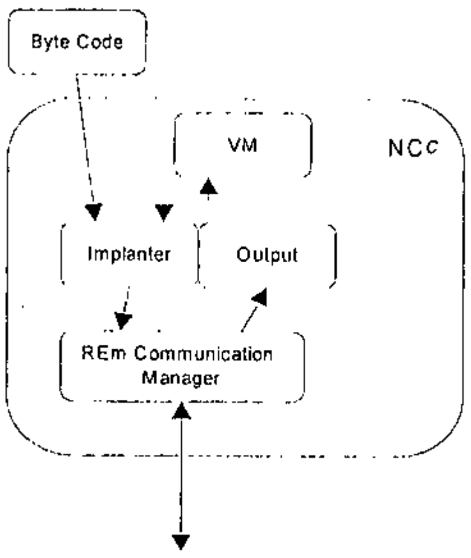


Fig (7): RE manager

5.5 Testing and measurements

We tested our simplified tools in two environments; over the Internet and within a local network. Two programs were used; a lightweight computation named ptest.p and a heavyweight computation name 1000.p. Since we can not control the network load at the time of experiments, we run the test four times, and calculated the average, in order to isolate the effect of transient heavy load of the network from the measurements.

Testing machines

Two machines were used for the test with the following configurations:

1- SUN: Sun Netra i5, 110MHz microSPARC II RISC, 64MB RAM, 16KB I-cache,8KB D-cache, running Solaris 2.6 and Java 1.1.

2- PC166: Pentium 166MHz, 64MB RAM, 512KB cache running Windows98, and Java 1.2.1.

3- PC333: Pentium II 333MHz, 64MB RAM, 128KB cache running Windows 95, and Java 1.2.1.

The network used is Ethernet running at 10MB/s with network protocol TCP/IP.

The Internet connection speed was 28.8 KB/s.

Testing configurations:

1- SUN: the SUN was used for both the client and the server. A session is created for each.

2- PC166: the PC166 was used for both the client and the server. A session is created for each.

3- PC333: the PC333 was used for both the client and the server. A session is created for each.

4- SUN-PC166: the SUN was used as a client and the PC166 is used as a server. Both are connected within a local network.

5- PC166-SUN: the PC166 was used as a client and the SUN was used as a server. Both are connected within a local network.

6- PC333-SUN: the PC333 was used as a client and the SUN was used as a server. Both were connected over the Internet.

Testing programs:

Two programs for testing was used one is ptest.p (Appendix C) which is a lightweight computation for calculating the factorial of 6, the other is 1000.p (Appendix C) which is a heavyweight computation that loops for ten million times.

Results of tests:

Table (4): results for testing the lightweight program ptest.p

Configuration/ Test Number	**SUN**	**PC166**	**SUN-PC166**	**PC166-SUN**	**PC333**	**PC333-SUN (IP)**	**PC333-SUN (DNS)**
1	1158	2580	1627	2030	6980	44540	37620
2	1000	2700	1629	1870	7690	38170	39650
3	988	2630	1641	1760	5820	41690	41960
4	992	3290	1584	1870	6530	36090	62950
Average	1034.5	2800	1620.25	1882.5	6755	40123	45545.5

Table (5): results for testing the heavyweight program 1000.p

Configuration **Test Number**	**SUN**	**PC166**	**SUN-PC166**	**PC166-SUN**
1	329972	92160	91071	310880
2	284943	91350	90863	296700
3	325681	92220	90049	326090
4	288757	91070	89799	285450
Average	307338.25	91700	90445.5	304780

Results analysis:

As shown in Table 4, the result for running the lightweight program on PC166 is faster than on PC333 and that is due to cache and windows version used. We noticed also that when running the program on SUN, the time it takes decreases every time approximately, this is due to the cache considerations. The effect of the Internet heavy load times can be seen on testing measurements in column PC333-SUN (DNS), test number 4, in which it is far above the average. In addition, we noticed that using the remote server IP address for doing the computation is faster than using its URL. That is because before any message transfer between the client and server the system asked the DNS for the IP address for the URL given.

Running the program on local slower machine (SUN) is better than using (SUN-PC166), i.e. it is not preferable to use the network to do computation for lightweight programs. On the contrary, as shown in Table 5, it is better to use the network while doing computation for heavyweight programs. That is because the server (PC166) is faster than the local machine and the time cost for communication is neglected compared to the performance gained.

Chapter 6

Conclusion and future work

6.1 Conclusion

Network computation is a promising area for information technology developers and users. It can add more power to existing applications and increase availability of systems for users.

Existing computation model classification is not satisfactory and for that, we developed a new classification based on the computation dimensions: code, data and processing such classification named CDP-Classification.

Existing computation models can not tackle the problem of legacy HLL programs. A new network computation model based on a combination of remote object execution (ROE), data fetch on demand (DOD) and remote code execution (RCE) of our CDP-Classification is introduced, developed and tested. Such model is called network computation model for high level languages (Implantation).

We believe that the Implantation can make use of computation power within a network best, that is because legacy HLL programs are implanted into the new environment without changing its semantic. At the same time, legacy HLL programs will use the network power transparent to the user. The model is transparent and failure resistant, unless the original node fails.

Some drawbacks are in the model, like size of messages between nodes, it is high compared to RPC, but most mobile code paradigms today have such a problem, Java applets for example.

Our model of implantation is portable regarding that the model does not depend on a specific machine design. Any language of implementation can be used, but we think Java is the best, since it facilitates portability and interoperability in an easy way. Java can easily make use of network programming regardless of the underling network protocol. That can be seen in the size of implementation code; the Java code for simplified NC*c* and NC*s* for example is around 1000 line.

6.2 Future Work

Implantation as we believe, can make use of the network best. Implantation model can be extended and used for future work as follows:

1- Security Model: In our work, we did not tackle the problem of network security feature. A robust security model can be added to the model. Jonathan (1999) discussed such issues for mobile code paradigm.

2- Rooming Agent: this can be easily implemented by modifying the HLL by adding constructs to inform the program about the environment it is running in (Implanted). such as where it is now and where to go.

3- Parallel programming can also be implemented by allowing the NC*c* to parallel execute the main program locally and at the same time send EU to other NC*s* to be executed with synchronization mechanism. Implementation can be easily achieved if Java programming language is used. Automatic code-analysis techniques must be used if we need to increase the level of transparence and automation for the execution of the implanted code.

4- We believe that designing a new paradigm as a hybrid from all paradigms is a new pioneered approach. It will be the solution of all computational problems in the future.

Appendix A

EBNF for Simple Pascal Language

program
    program-heading block "."

program-heading
    program identifier ";"

block
    declaration-part statement-part

declaration-part
    [type-definition-part]
    [variable-declaration-part]
    procedure-and-function-declaration-part

type-definition-part
    type type-definition ";" { type-definition ";" }

type-definition
    identifier "=" type

variable-declaration-part
    var variable-declaration ";" { variable-declaration ";" }

variable-declaration
    identifier-list ":" type

procedure-and-function-declaration-part
    { (procedure-declaration | function-declaration) ";" }

procedure-declaration
    procedure-heading ";" procedure-body

procedure-identification ";" procedure-body

procedure-body
    block

function-declaration
    function-heading ";" function-body

function-identification ";" function-body

function-body
    block

statement-part
  begin statement-sequence **end**

Procedure and Function Definitions

procedure-heading
  procedure identifier [formal-parameter-list]

function-heading
  function identifier [formal-parameter-list] ":" result-type

result-type
  type-identifier

procedure-identification
  procedure procedure-identifier

function-identification
  function function-identifier

formal-parameter-list
  "(" formal-parameter-section { ";" formal-parameter-section } ")"

formal-parameter-section
  value-parameter-section |
  variable-parameter-section |
  procedure-parameter-section |
  function-parameter-section

value-parameter-section
  identifier-list ":" parameter-type

variable-parameter-section
  var identifier-list ":" parameter-type

procedure-parameter-section
  procedure-heading

function-parameter-section
  function-heading

parameter-type
  type-identifier | conformant-array-schema

conformant-array-schema
  array-schema

array-schema
    array "[" bound-specification { ";" bound-specification } "]"
    of (type-identifier | conformant-array-schema)

bound-specification
    identifier ".." identifier ":" ordinal-type-identifier

ordinal-type-identifier
    type-identifier

Statements

statement-sequence
    statement { ";" statement }

statement
    (simple-statement | structured-statement)

simple-statement
    [assignment-statement | procedure-statement]

assignment-statement
    (variable | function-identifier) ":=" expression

procedure-statement
    procedure-identifier [actual-parameter-list]

structured-statement
    compound-statement | repetitive-statement | conditional-statement | with-statement

compound-statement
    begin statement-sequence **end**

repetitive-statement
    while-statement | repeat-statement

while-statement
    while expression **do** statement

repeat-statement
    repeat statement-sequence **until** expression

initial-expression
    expression

final-expression
    expression

conditional-statement
    if-statement | case-statement

if-statement
    if expression **then** statement [**else** statement]

actual-parameter-list
    "(" actual-parameter { "," actual-parameter } ")"

actual-parameter
    actual-value | actual-variable | actual-procedure | actual-function

actual-value
    expression

actual-procedure
    procedure-identifier

actual-function
    function-identifier

Expressions

expression
    simple-expression [relational-operator simple-expression]

simple-expression
    [sign] term { addition-operator term }

term
    factor { multiplication-operator factor }

factor
    variable | number | string | constant-identifier | bound-identifier | function-designator | "(" expression ")"

relational-operator
    "=" | "<>" | "<" | "<=" | ">" | ">="

addition-operator
    "+" | "-"

multiplication-operator
    "*" | "/" | div

variable
    entire-variable | component-variable | referenced-variable

entire-variable
    variable-identifier | field-identifier

component-variable
    indexed-variable | field-designator | file-buffer

indexed-variable
    array-variable "[" expression-list "]"

field-designator
    record-variable "." field-identifier

function-designator
    function-identifier [actual-parameter-list]

Types

type
    simple-type | structured-type | type-identifier

simple-type
    subrange-type

subrange-type
    lower-bound ".." upper-bound

lower-bound
    constant

upper-bound
    constant

structured-type
    unpacked-structured-type

unpacked-structured-type
    array-type | record-type

array-type
    array "[" index-type { "," index-type } "]" **of** element-type

index-type
    simple-type

element-type
    type

record-type
    record field-list end

Record Fields

field-list
    [(fixed-part [";"]]

fixed-part
    record-section { ";" record-section }

record-section
    identifier-list ":" type

Variable and Identifier Categories

identifier
    letter { letter | digit }
file-variable
    variable
record-variable
    variable
actual-variable
    variable
array-variable
    variable
field-identifier
    identifier
constant-identifier
    identifier
variable-identifier
    identifier
type-identifier
    identifier
procedure-identifier
    identifier
function-identifier
    identifier
bound-identifier
    identifier

variable-list
    variable { "," variable }
identifier-list
    identifier { "," identifier }
expression-list
    expression { "," expression }
number
    integer-number | real-number
integer-number

```
    digit-sequence
real-number
    digit-sequence "." [ digit-sequence ] [ scale-factor ] |
    digit-sequence scale-factor
scale-factor
    ("E" | "e") [ sign ] digit-sequence
unsigned-digit-sequence
    digit { digit }
digit-sequence
    [ sign ] unsigned-digit-sequence
sign
    "+" | "-"
letter
    "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" | "N" | "O" |
"P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" |
    "X" | "Y" | "Z" | "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" |
"n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" |
    "x" | "y" | "z"
digit
    "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

Appendix B

Basic Instruction Set for the Stack Machine

Mnemonic	Parameter	Action
0 pushcI	CI	SP := SP + 1; Stack[SP] := CI;
1 pushI	O	SP := SP + 1; Stack[SP] := Stack[FP+O];
2 popI	O	Stack[FP+O] := Stack[SP]; SP := SP - 1;
3 pushgI	N, O	SP := SP + 1; Stack[SP] := Stack[base(N)+O];
4 popgI	N, O	Stack[base(N,FP)+O] := Stack[SP]; SP := SP - 1;
5 fetchI		Stack[SP] := Stack[Stack[SP]];
6 popiI		Stack[Stack[SP-1]] := Stack[SP]; SP := SP - 2;
7 pusha	O	SP := SP + 1; Stack[SP] := FP+O;
8 pushga	N, O	SP := SP + 1; Stack[SP] := base(N,FP)+O;
33 pushs	CS	SP := SP + 1; Stack[SP] := &CS;
39 storeI		Stack[Stack[SP-1]] := Stack[SP]; Stack[SP-1] := Stack[SP]; SP:= SP - 1;
9 addI		Stack[SP-1] := Stack[SP-1] + Stack[SP]; SP := SP - 1;
10 subI		Stack[SP-1] := Stack[SP-1] - Stack[SP]; SP := SP - 1;
11 mulI		Stack[SP-1] := Stack[SP-1] * Stack[SP]; SP := SP - 1;
12 divI		Stack[SP-1] := Stack[SP-1] div Stack[SP]; SP := SP - 1;
13 negI		Stack[SP] := - Stack[SP]
16 eqI		Stack[SP-1] := ord(Stack[SP-1] = Stack[SP]); SP := SP - 1;
17 neI		Stack[SP-1] := ord(Stack[SP-1] <> Stack[SP]); SP := SP - 1;

18 ltI		Stack[SP-1] := ord(Stack[SP-1] < Stack[SP]); SP := SP - 1;
19 leI		Stack[SP-1] := ord(Stack[SP-1] <= Stack[SP]); SP := SP - 1;
20 gtI		Stack[SP-1] := ord(Stack[SP-1] > Stack[SP]); SP := SP - 1;
21 geI		Stack[SP-1] := ord(Stack[SP-1] >= Stack[SP]); SP := SP - 1;
22 jumpz	L	if Stack[SP] = 0 then PC := &L; SP := SP - 1;
23 jumpnz	L	if Stack[SP] < > 0 then PC := &L; SP := SP - 1;
24 jump	L	PC := &L;
25 enter	N	Stack[SP+1] := base(N,FP); Stack[SP+2] := FP; SP := SP + 5;
26 alloc	N	SP := SP + N;
36 setrvI		Stack[FP] := Stack[SP]; SP := SP - 1;
27 return		SP := FP - 1; PC := Stack[FP+4]; FP := Stack[FP+1];
35 returnf		SP := FP; PC := Stack[FP+2]; FP := Stack[FP+1];
28 call	L, N	FP := SP - (N+4); Stack[FP+4] := PC; PC := &L;
14 int		Stack[SP] := trunc(Stack[SP]);
15 intb		Stack[SP-1] := trunc(Stack[SP-1]);
142 flt		Stack[SP] := float(Stack[SP]);
143 fltb		Stack[SP-1] := float(Stack[SP-1])
255 stop		halt;

where base(N,FP) = FP if (N=0)

= base(N-1,stack[FP+1]) if (N!=0)

Appendix C

Test program Pptest.p

```
program main;
 var result:integer;
  procedure fact(n:integer);
   var i,r,p : integer;

   procedure mul(n1:integer;n2:integer);
   begin
     p:=n1*n2;
   end;

  begin
   i:=1;
   r:=1;
   while (i<=n) do
     begin
       mul(r,i);
       r:=p;
       i:=i+1;
     end;
   result:=r;
  end;

begin
  fact(5);
end.
```

Assembly code for Pptest

```
$mul2
          alloc    0
          pushga 1          ,8
          pusha  5
          fetchI
          pusha  6
          fetchI
          mulI
          popiI
          return
$fact1
          alloc    3
          pusha  6
          pushcI 1
          popiI
          pusha  7
```

```
        pushcl 1
        popil
$3
        pusha  6
        fetchl
        pusha  5
        fetchl
        lel
        jumpz $4
        enter 0
        pusha  7
        fetchl
        pusha  6
        fetchl
        call      $mul2 ,2
        pusha  7
        pusha  8
        fetchl
        popil
        pusha  6
        pusha  6
        fetchl
        pushcl  1
        addl
        popil
        jump $3
$4
        pushga 1          ,5
        pusha  7
        fetchl
        popil
        return
main
        enter 0
        alloc    1
        enter 0
        pushcl  5
        call      $fact1 ,1
        stop
```

Test program ptest.p

```
program s;
  var res : integer;
  procedure f(n:integer);
    var i,r : integer;
  begin
    i:=1;
    r:=1;
    while (i<=n) do
      begin
        r:=r*i;
        i:=i+1;
      end;
    res:=r;
  end;
begin
  f(6);
end.
```

Assembly code for ptest.p

```
$f1
        alloc   2
        pusha   6
        pushcI  1
        popiI
        pusha   7
        pushcI  1
        popiI
$2
        pusha   6
        fetchI
        pusha   5
        fetchI
        leI
        jumpz $3
        pusha   7
        pusha   7
        fetchI
        pusha   6
        fetchI
        mulI
        popiI
        pusha   6
        pusha   6
        fetchI
        pushcI  1
        addI
        popiI
        jump $2
$3
        pushga 1        ,5
        pusha  7
        fetchI
        popiI
        return
main
        enter 0
        alloc    1
        enter 0
        pushcI 6
        call      $f1 ,1
        stop
```

Test program 1000.p

```
program s;
  var res: integer;
  procedure loop;
    var i,j,k,r : integer;
  begin
    i:=0;
    while (i<10) do
      begin
       j:=0;
        while (j<1000) do
          begin
            k:=0;
            while (k<1000) do
              begin
                k:=k+1;
              end;
            j:=j+1;
          end;
         i:=i+1;
      end;
      res:=i;
   end;
begin
  loop;
end.
```

Assembly code for 1000.p

```
$loop1
        alloc   4
        pusha   5
        pushcl  0
        popil
$2
        pusha   5
        fetchl
        pushcl  10
        ltl
        jumpz $3
        pusha   6
        pushcl  0
        popil
$4
```

```
        pusha  6
        fetchI
        pushcI 1000
        ltI
        jumpz $5
        pusha  7
        pushcI 0
        popiI
$6
        pusha  7
        fetchI
        pushcI 1000
        ltI
        jumpz $7
        pusha  7
        pusha  7
        fetchI
        pushcI 1
        addI
        popiI
        jump $6
$7
        pusha  6
        pusha  6
        fetchI
        pushcI 1
        addI
        popiI
        jump $4
$5
        pusha  5
        pusha  5
        fetchI
        pushcI 1
        addI
        popiI
        jump $2
$3
        pushga 1          ,5
        pusha  5
        fetchI
        popiI
        return
main
        enter 0
        alloc    1
        enter 0
        call $loop1 ,0
        stop
```

Appendix D

Table of Symbol used

Symbol	meaning
API	Application Program Interface
CORBA	Common Object Request Broker Architecture
DCOM	Distributed Component Object Model
DNS	Distributed Name Service
EU	Execution Unit (Procedure, Function, Subroutine, ... etc)
EULT	Execution Unite Lookup Table
HLL	High Level Languages like (Pascal, FORTRAN ... etc)
MAP	Mobile Agent Platform
NC	Network Computation
NCc	Network Computation Client
NCM	Network Computation Model
NCN	Network Computation Node
NCNet	Network Computation Network
NC*r*	Network Computation Registry
NC*s*	Network Computation Server
PLAN	Programming Language for Active Networks
PVM	Parallel Virtual Machine
RE manager	Simplified NC*c* : Remote Execution Manager
RMI	Remote Method Invocation
RPC	Remote Procedure Call
TCP/IP	Transmission Control Protocol / Internet Protocol
UGAVAC	A stack oriented machine named UGAVAC
URL	Universal Resource Locator
VM	Virtual Machine
VMServer	Simplified NC*s* : Virtual Machine Server

References

Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek and Vaidy Sunderam, *"PVM: Parallel Virtual Machine: user guide and tutorial for networked parallel computing"* MIT Press, London, 1994.

Alice Fischer, Frances Grodzinsky, *"The Anatomy of Programming Languages"*, Printice Hall, 1993.

Andrw S. Tanenbaum, *"Distributed Operating Systems"*, Prentice Hall, 1995.

Anthony D. Joseph, Joshua A. Tauber and M. Frans Kaashoek, "Mobile Computing with the Rover Toolkit", IEEE Transactions on computers, Vol 46, No 3: March 1997.

Antonio Carzaniga, Gian Pietro Picco and Giovanni Vigna, *"designing distributed application with mobile code paradigms"*, 1997.

Arthur Dumas, *"Programming WinSock"*, Sams publishing, 1995.

Barry Wilkinson, *"Computer Architecture : design and performance"* 2nd edition, Prentice Hall, 1996, P225-441.

Bibliography on Software Agents1, Heikki Helin ,University of Helsinki, department of Computer Science, May 18, 1999 http://www.cs.helsinki.fi/~hhelin/agents/agent-bib.html.

Cornell Theory Center, *"Tools for Source code analysis "*,

http://www.tc.cornell.edu/Parallel.Tools/src-analysis-tools.html

Doreen Y. Cheng, *"A Survey of Parallel Programming Languages and Tools"*, NASA Ames Research Center, 1993.

Elliotte Rusty Harold, *"Java Network Programming"*, O'REILLY, 1997.

Fred Halsall, *"Data Communications, Computer Networks and Open Systems"*, Addison-Wesley, 1996.

G. Adorni and A. Pogg, *"MAP: A language for modeling multiple agent systems"*, 1993.

George Necula and Peter Lee, *"Research on Proof-Carrying Code of Untrusted-Code Security"*, 1997.

George R. Desrochers, *"Principles of Parallel and Multiprocessing"*, McGraw-Hill, 1987.

Gerard Holzmann and Björn Pehrson, *"The Early History of Data Networks Napoleon's Internet"*, 1994.

Henri E. Bal , *" Report on the Programming Language Orca"*, Dept. of Math. And Computer Science, Vrije Universiteit ,1994

Henri E. Bal, *"A comparative study of five parallel programming languages"*, Vrije Universiteit amsterdam, 1991.

James Gosling , Bill Joy and Guy Steele, *"The Java Language Specification"*, Addison-Wesley, 1996

Jim Farley, *"Java Distributed Computing"*, O'REILLY , 1998.

Jonathan T. Moore, *"Mobile Code Security Techniques"*, University of Pennsylvania, 1999.

Kristian Paul Bubendorfer, *"Resource Based Policies for Load Distribution"*, Thesis, Victoria University of Wellington, 1996.

L. V. Kale, *"The Charm Parallel Programming Language"*, University of Illinois, 1994.

Michael Hicks and Angelos D. Keromytis , *"A Secure Plan"*, University of Pennsylvania, 1999.

Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles, *"Network Programming using PLAN"*, university of Pennsylvania, 1998.

Open Group , *"Network Computer"*, C720 February 1998.

P. Emerald Chung, Yannun Huang, Shalini Yajnik, Deron Liang, Joanne C. Shih, Chung-Yih Wang and Yi-Min Wang, *"DCOM and CORBA Side by Side, Step by Step, and Layer by Layer"*, AT&T Labs, 1997.

Pankaj Kakkar, *"The Specification of PLAN (Draft1)"*, DARPA , July 1999.

Peter Domel, *"Mobile Telescript and the web"*, IEEE computer society press, 1996.

Rajive Bagrodia and Wen-Toh Liao, *"Maisie User Manual, Rel 2.2"*, University of California, 1995.

Renhard Wilhelm and Dieter Maurer, *"Compiler Design"*, Addison-Wesley , 1995.

Robert Macgregor, Dave Durbin John Owlett and Adrew Yeomans, *"Java Network Security"*, Prentice-Hall,1998.

Russian Academy of Sciences, *"The mpC programming Language specification"*, 1997.

Sun Systems, *"The Java Virtual Machine Specification"* Rel 1.0 Sun Systems, 1995.

Tommy Thorn, *"Programming Languages for Mobile Code"*, ACM Computing Surveys, Vol. 29, No. 3. September 1997.

ملخص

أدوات برمجية لرفع قدرة لغات البرمجة العالية لاستخدام ميزة العمل على الشبكات

إعداد

عبدالله محمد ابوعياش

إشراف

الدكتور رياض جبري

منذ مدة طويلة يتم البحث في مجال الحوسبة بواسطة الشبكات أو ما يسمى بحوسبة الشبكات، حيث يعد هذا المجال من اكثر المجالات الحديثة اهتماما وخصوصا من قبل مطوري ومستعملي تكنولوجيا المعلومات، وتعني كلمة حوسبة الشبكات استخدام الشبكات أثناء تنفيذ الأنظمة والبرامج، ومن الدلائل على ذلك انتشار اســـــتخدام الإنـــترنت وتطبيقاتها، في هذه الايام يتم استعمال طرقة مختلفة لحوسبة الشبكات، الا ان كل الطرق لا تتطرق الى اســـــتخدام البرامج المتوارثة والمكتوبة بواسطة لغات البرمجة العالية. في هذه الرسالة تم دراسة نماذج استخدام حوسبة الشبكات. تم اقتراح تصنيف جديد لحوسبة الشبكات واقتراح نموذج جديد لاستخدام الانظمة المتوارثــــة وقـــد تم تســـميت النموذج الجديد "الزراعة" Implantation.

إن اهتمامنا في هذه الدراسة هو في بحث بعض النماذج المستخدمة حالية في حوسبة الشبكات وما هـــي ميزاتهـــا وتصنيفاتها وتطبيقاتها وبعض لغات البرمجة والأدوات المستخدمة فيها. وسنقدم في هذه الدراسة تصنيفـــــا جديـــدا لتصنيف حوسبة الشبكات وقد اسميناه CDP-Classification.

لقد تم تطوير النموذج الجديد من خلال استخدام الجهاز الافتراضي للغات البرمجة العالية ، حيث سيتم "زرع" أجزاء من هذه الأنظمة في أجهزة متصلة مع بعضها البعض بشبكة حاسوب لكي تعمل عليها، ، إن هذا الأسلوب يمكـــن الأنظمة المتوارثة من العمل في بيئة الشبكات.