A .Net Based Resource Sharing Framework

by

Xiaohan Lin

Submitted to the Department of Civil and Environmental Engineering
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in The Field of Information Technology

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

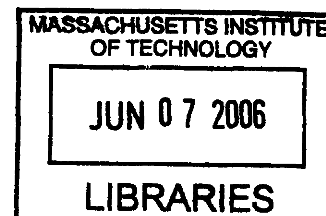
June 2006

© Massachusetts Institute of Technology 2006. All rights reserved.

Author ....
Department of Civil and Environmental Engineering
May 8, 2006

Certified by ....
John R. Williams
Associate Professor of Civil and Environmental Engineering
Thesis Supervisor

Accepted by ....
Andrew Whittle
Chairman, Department Committee for Graduate Students


MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JUN 0 7 2006
LIBRARIES

ARCHIVES

A .Net Based Resource Sharing Framework

by

Xiaohan Lin

Submitted to the Department of Civil and Environmental Engineering
on May 8, 2006, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in The Field of Information Technology

Abstract

This thesis presents an *Internet resource sharing architecture*. It allows users to access and utilize unused computer resources, such as CPU cycles and storage, without an expert's knowledge. It achieves this by providing a number of abstract services that hide some of the complexity inherent in distributed computing. In recent years, Grid Computing has been proposed as a solution for Internet resource sharing. However, Grid Computing as presently implemented does not address the need of the large majority of the users. In this thesis, we propose a different approach to achieve Internet resource sharing called the *Realm*.

The *Realm Framework* offers a lightweight layer on top of the Microsoft .Net Framework so that the programs that can be migrated to .Net Framework can also utilize the shared resources through the Realm Framework. By leveraging the Microsoft .Net Framework, the Realm Framework avoids tedious re-working in this fast-paced world of technology by sitting on the top of the full-featured, coherent and up-to-date development platform. The Realm Framework applies current technologies such as Web Services, the Common Language Runtime (CLR) and popular encryption algorithms.

In this thesis a versatile runtime system and a set of extension interfaces in C# programming language is developed. The modularized software package offers a layered programming model for distributed-application developers with different levels of proficiency. Two utilities that are helpful for maintaining a distributed system are also developed, namely, a dynamic domain-name based inter-realm communication scheme and a distributed debugger.

Examples of applying the Realm Framework to several typical scenarios are shown, including embarrassingly parallel problems that require little communication between computing nodes, parallel computing problems that require intensive message-passing between the computing nodes, and universal storage systems that are based on storage media and the messenger-like applications that require a sophisticated communication scheme.

Thesis Supervisor: John R. Williams

Title: Associate Professor of Civil and Environmental Engineering

Acknowledgments

First, I am very grateful to Professor John R. Williams for his kind support, endless curiosity, and contagious excitement throughout my 5 years at MIT; for his continuous encouragement on my ideas and projects; and for his always taking time out of his busy schedule to discuss any problems, including personal suggestions and support during a legal issue. It has been a great pleasure to work as a student under his supervising.

Thanks to my PhD committee, who made this thesis possible. Thanks to Professor Steven Lerman and Dr. Judson Harward, for their curiosity and valuable suggestions to my research.

I would also like to thank Professor Feniosky Pena-Mora. I learned a great deal from him during my first year of study at MIT.

Also, thanks to my friends at MIT, especially to the lab-mates of the Intelligent Engineering Systems Laboratory (IESL). They made MIT fun to stay.

Special thanks to Joan McCusker, and other administrative staff in CEE department. Their help and advice made my study at MIT a smooth experience.

And of course, my lovely wife, for her patience and understanding during the last year, my busiest year, my first year of marriage. Thank you for offering the spiritual support.

Lastly, I would like to thank to my parents, who stay far away at the other end of the earth. It is the time to reward you, my dad and mom, for making all these things possible.

Contents

1 Introduction **15**

1.1 Managed under The Digital Kingdom 15

1.2 Thesis Goals 17

1.3 Thesis Outline 17

2 Historical Review and The Realm Idea **21**

2.1 Computer Clusters vs. Integrated Supercomputers 21

2.2 Ancestors and Siblings of The Realm Framework 23

2.2.1 The Old-school Systems 23

2.2.2 Recent Efforts toward Modern Internet Resource-Sharing . . . 25

2.3 The Realm Framework 28

2.3.1 Potential Application Scenerios 29

2.3.2 Features to be Implemented 35

2.4 Research Topics and Challenges 36

2.4.1 Accessibility 36

2.4.2 Programmability 37

2.4.3 Reliability 38

2.4.4 Efficiency 39

2.5 Context and Conclusion 39

3 The Realm Runtime System **41**

3.1 The Development Platform: .Net Framework and C# 41

3.1.1 Limitations of the Traditional Systems 42

3.1.2 Advantages of Using the .Net Framework and C# 43
3.2 The Realm Distributed Resource Sharing Runtime System 44
3.2.1 Goals of Design 44
3.2.2 System Design 45
3.2.3 Process Management 47
3.2.4 Intercommunication between the Worker Processes 49
3.3 Accessibility 51
3.3.1 Realm Setup 51
3.3.2 Seed Programming 53
3.3.3 Seed Debugging 55
3.4 Programmability 55
3.4.1 The Bottom Level Interface 56
3.4.2 The Parallel Computing Interfaces 58
3.4.3 The Universal Object Storage Interfaces 62
3.4.4 The Messenger Interface 63
3.5 Reliability 64
3.5.1 Security 64
3.5.2 Fault-Tolerance and Adaptive Parallelism for Embarrassingly Parallel Applications 70
3.6 Efficiency 72
3.6.1 Runtime Efficiency 72
3.6.2 Communication Efficiency 74
3.7 Conclusion 74

4 The Embarrassingly Parallel Applications 75
4.1 Typical Network Setup 75
4.2 64 bit RSA5 Encryption Cracking 76
4.3 Movie Making Using POV-Ray 79

5 The Communication-intensive Parallel Applications 83
5.1 Typical Network Setup 83

5.2 Simulate the Mixture of Fine Solid Particles and Fluid Using Lattice-Boltzmann Method 84
5.2.1 The Lattice-Boltzmann Method 84
5.2.2 Simulate the Fine Particles 86
5.2.3 Domain Decomposition 87
5.2.4 Results 88
5.3 Distributed Discrete Element Method 92
5.3.1 Domain Partitioning 93
5.3.2 Results and Comparisons 97

6 The Storage Applications 99
6.1 Typical Network Setup 99
6.2 Saving Data on The Local File System 100
6.3 Save Data into Gmail Account 100
6.4 A Virtual File Browser for the Universal Object Storage 102

7 The Messenger Application 103
7.1 Problem Introduction 103
7.2 A Simple Supply Chain Communication System 104

8 Supporting Components 109
8.1 The Dynamic DNS System 109
8.2 The Realm Debugger 111
8.3 Limitations 112

9 Conclusion 115
9.1 Summary 115
9.2 Contributions 116
9.3 Future Work 117
9.4 Final Words 119

List of Figures

2-1 Parallel systems since 1993. 22

2-2 Microsoft HPC network (from Microsoft). 24

2-3 The simple Master-Slave style parallel computing topology. 30

2-4 The MPI-style parallel computing topology. 32

2-5 The interaction between a Universal Object Storage node with other nodes. 33

2-6 The channeled communication style. 34

3-1 Components in the Realm Framework. 45

3-2 Structure of a Realm worker. 46

3-3 Creation of the Worker processes. 48

3-4 Runtime control components. 49

3-5 Communication scheme. 51

3-6 The WorkerI interface (Worker manager on Windows). 54

3-7 The WorkerI interface (Worker manager on Windows). 56

3-8 The .Net Remoting message chains (from Microsoft). 66

3-9 The .Net Remoting message with authenticaion in SOAP format. . . 67

3-10 The .Net Remoting negotiating step. 69

3-11 The secured .Net Remoting. 69

3-12 The default scheduler used in the Realm system. 71

3-13 Comparison of efficiency. 73

4-1 The typical network setup for the master-slave style parallel computing. 76

4-2 Performance for different partition size. 78

4-3 Performance for different network setup (partition size=2^{16}). 78
4-4 Performance for different network setup (partition size=2). 79
4-5 Movie frames generated by the distributed POV-Ray based on the Realm Framework. 81

5-1 Performance for different network setup (partition size=2. 87
5-2 2D Lid-driven flow. 88
5-3 Blood flow in a branched blood vessel. 89
5-4 Thick mixture in a blender. 90
5-5 Particles accumulating around a hole. 90
5-6 Convergence of the time cost. 91
5-7 The 1D global partitioning using a reference axis. 96
5-8 Spheres in a cynlinder (512 spheres). 97

6-1 Gmail account used for Universal Object Storage. 101
6-2 The virtual file browser. 102

7-1 The parties in a supply chain. 105
7-2 Supply chain emulator interfaces. 107

8-1 The Realm Dynamic DNS System. 111
8-2 The Realm Debugger interface. 113

List of Tables

5.1 Communication Channel performance comparison (Blender, 8 nodes). 92

5.2 DEM Parallel Partitioning performance comparison (2048 spheres, 8 nodes). 98

Chapter 1

Introduction

1.1 Managed under The Digital Kingdom

In English literature, the word *Realm* originally meant *"A royal jurisdiction or domain; a region which is under the dominion of a king; a kingdom."*[1] Today, a Realm is primarily a synonym for a world other than our own. The word *Realm* is often used in fantasy books or movies. It is also seen in some technical fields. For example, in Java EE, Realm refers to a database containing users, usergroups and their roles [2]. Optionally a Realm manages user-passwords, certificates and authentication logic. Also a Realm can refer to a Web domain. A common feature of this term under all these contexts is that it describes a populated entity that is well under control, composed of multiple function units, and more importantly, accessible [3] by some particular protocols.

This thesis seeks to bring the advantages enjoyed by a physical Realm of individuals into the world of computing by presenting and developing the *Realm Framework*, that makes it easy for computer users on the Internet to share their computer resources in a coordinated and secure manner. Within the computational Realm, processes are able to 1) work together to solve large-scale computational problems that are not possible for any single machine to handle. 2) put together different types of storage

[1] Webster's Revised Unabridged Dictionary (1913).

[2] The roles are sets of permissions to access server-resources.

[3] Both internally among the components and externally to the guests.

that can be accessed by a common protocol, and 3) talk to neighbors in a reliable manner.

The Realm Framework allows ordinary computer users, especially those using Microsoft Windows to share their idle processing powers (CPU cycles) or their storage space without help from an expert. On the other hand, *Realm programmers* can write software programs for the Realm Framework so that a Realm can solve computational problems by taking advantage of the shared resources. A Realm is simply a distributed computing system. Unlike its predecessors, such as MPI and Grid Computing, however, the Realm Framework offers easy installation, good programmability, a standard communication interface and several other features that will be discussed in this thesis.

By making it easy for people to share their computers' idle processing power or storage space, the Realm Framework creates many new possibilities. Since the majority of computers are running Microsoft Windows Operating Systems, the Realm potentially has a large supply of resources that it can consolidate. This is a significant advantage over the traditional parallel systems that are usually running on clusters of Unix machines. Moreover, because it allows inexperienced computer users to share their resources, the Realm Framework can be used to build extremely large computing networks without investing in expensive hardware. We have tested an 1024-node Realm without crashing the system. The number of nodes supported can potentially be much larger. With the help of the inter-Realm communication facility discussed in Chapter 8, there can be an unlimited number of computers in the *Realm world.* This offers affordable solutions to most companies and organizations, even in developing countries.

In fact, the Realm Framework has been successfully used to solve a number of problems in engineering [23, 36, 38]. One problem simulates the dynamics of an oil-sand mixture. The simulator was based on the Lattice-Boltzmann method [6, 28] and a simplified version of the Discrete Element Method (DEM [8]). In a typical execution, the Realm utilized eight computers and successfully finished the simulation after three days and transferred around 1.2 gigabytes of data. Another simulator was

also developed using the Realm Framework and the Discrete Element Method. In a subsequent effort we have successfully migrated Scott's Johnsons PhD code [21] to the distributed environment and made it a convenient simulation package in which end-users are virtually unaware of the existence of the Realm that supplies automatic partitioning and job scheduling. The thesis describes these applications in Chapters 4 to 7.

1.2 Thesis Goals

Distributed computing is not a new topic. However, with the development of new technologies, such as XML, Web services and Common Language Runtime (CLR) in recent years, it needs a renew effort to explore the novel possibilities that a distributed computing system can bring. This thesis proposes the idea of the Realm Framework and the detailed implementation of the runtime system. We show that the Realm Framework goes beyond the traditional approach of managing distributed resources by demonstrating that:

1. it is easily accessible for both end-users and Realm programmers, and thus facilitates the building of large scale computing networks,
2. it is reliable even in a potentially hostile, open network,
3. it is versatile and can be adapted to many application scenarios,
4. it is extensible.

We compare the Realm Framework with its predecessors and siblings. We show that it is a useful alternative of MPI, Grid Computing and other distributed systems when solving some problems. We also note some disadvantages of the Realm system.

1.3 Thesis Outline

The thesis first presents the idea of the resource sharing framework, then the implementation of the Realm runtime system. The following chapters discuss use cases in

four distinct categories. We emphasize the sharing of processing power sharing because parallel computing is the most obvious useful case. The rest of the thesis deals with those miscellaneous topics that do not fit naturally into the former chapters.

Historical Review and The Realm Idea

Chapter 2 presents the motivations of this study. We first highlight the current trend that the supercomputing world favors computer clusters instead of integrated supercomputers. We follow this trend and briefly review various systems that have a similar design. The Realm idea is then proposed as an alternative to the other systems to fit the need of today's users. Research topics and challenges are identified, and the remainder of the thesis addresses their solutions.

The Realm Runtime System

We first explain why we choose the .Net Framework and C# programming language as the development platform. Then a general architecture of the Realm Framework is given. The research topics mentioned in Chapter 2, mainly accessibility, programmability, reliability and efficiency, are discussed in detail.

Four Typical Use Cases of The Realm Framework

Chapter 4, 5, 6 and 7 focus on application programming and performance comparisons. We demonstrate how the Realm Framework can be used in four typical scenarios: embarrassingly parallel computing[4], communication-intensive parallel computing, Universal Object Storage and messenger-like applications. We develop them to show the capability of the Realm system.

[4]The "embarrassingly parallel computing" is a term referring to those problems that can be partitioned into smaller independent jobs.

Supporting Components

Two very useful tools are furnished by the Realm system. The dynamic domain name system enables the user to identify and locate a resource by using a fixed hostname; the Realm debugger helps the application programmer develop Realm-based applications more quickly. They are discussed in Chapter 8.

Chapter 2

Historical Review and The Realm Idea

In this chapter, we review the previous work and current efforts for sharing computational resources over a computer network. We also identify the requirements and desired features that current Internet users would expect from the resource-sharing system. With these new requirements in mind, we proceed to identify the research issues and challenges we are facing when developing a new resource-sharing framework. In general, this chapter *raises the problems* to be solved by the rest of the thesis.

2.1 Computer Clusters vs. Integrated Supercomputers

During the past ten years, there has been a trend in the Supercomputing world–more and more people are choosing multiple clustered machines instead of one single mainframe supercomputer. Figure 2-1 shows the result of a survey by TOP500 [1] This chart illustrates the architectures/systems that people have been using since 1993. For each year, the top 500 super computing systems are categorized into six different architectures denoted by different grey levels. Before 1998, there were few practical

[1] See web site http://www.top500.org/

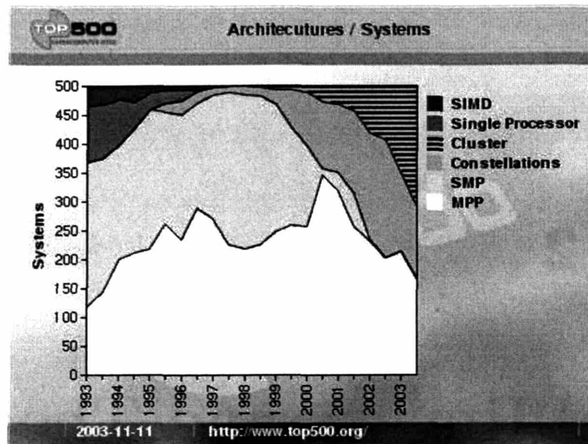


Figure 2-1: Parallel systems since 1993.

clustered systems, while in 2003, almost half of the top 500 super computer systems reported were computer clusters. The attraction of clusters lies in the (potentially) low cost of both hardware and software and the control that builders/users have to scale-up their systems to meet demand. There is generally a large difference in the usage of clusters and their more integrated counterparts: clusters are mostly used for capability computing while the integrated machines primarily are used for capacity computing[34]. In capability computing the system is employed for one or a few programs for which no alternative is readily available in terms of computational capabilities. In capacity computing, the system is employed to the full by using most of its available cycles for many, often very demanding, applications and users. However, as clusters become on average both larger and more stable, there is a trend to use them also for capacity computing, which has been dominated by integrated machines [34].

2.2 Ancestors and Siblings of The Realm Framework

2.2.1 The Old-school Systems

SETI@Home is one of the early efforts to share CPU cycles across the Internet. SETI@home was originally developed to analyze radio telescope signals by connecting and using computers ("volunteers") in homes and offices around the world. This approach, though it presented some difficulties, has provided significant computing power and has led to a unique public involvement in science. SETI@Home targets embarrassingly parallel problems only, and their efficiency of computing but not portability or standard compliance. Besides, there was no differentiation of "roles", such as "computing facility" and "storage", among volunteer computers since SETE@Home only shares idle processing power.

An almost identical system for embarrassingly parallel problems is the *distributed.net* [17]. Instead of being dedicated to just one project as SETI@Home does, distributed.net is the Internet's first *general-purpose* distributed computing system. It has successfully done some truly large-scale supercomputing jobs, such as cipher challenges and Optimal Mark Golomb Rulers [16]. In spite of these significant achievements, distributed.net, as well as SETI@Home, are not capable of handling problems that require communication between worker nodes. They do not offer coordination of node-to-node messaging which is critical in a *true* parallel computing system.

The Message Passing Interface (MPI) standard is the most commonly used parallel computing framework for computer clusters. It goes well beyond the simple master-slave topology demonstrated by SETI@Home and distributed.net. The history of MPI dates back to 1992, when a common standard for parallel computing was proposed [13]. Most MPI implementations have been written for Unix platforms which provide convenient tools and libraries for process control, networking and storage. In practice, each Unix machine is a computing worker. They are inter-linked by a high-speed network connection. They usually share files through a Network File System (NFS)

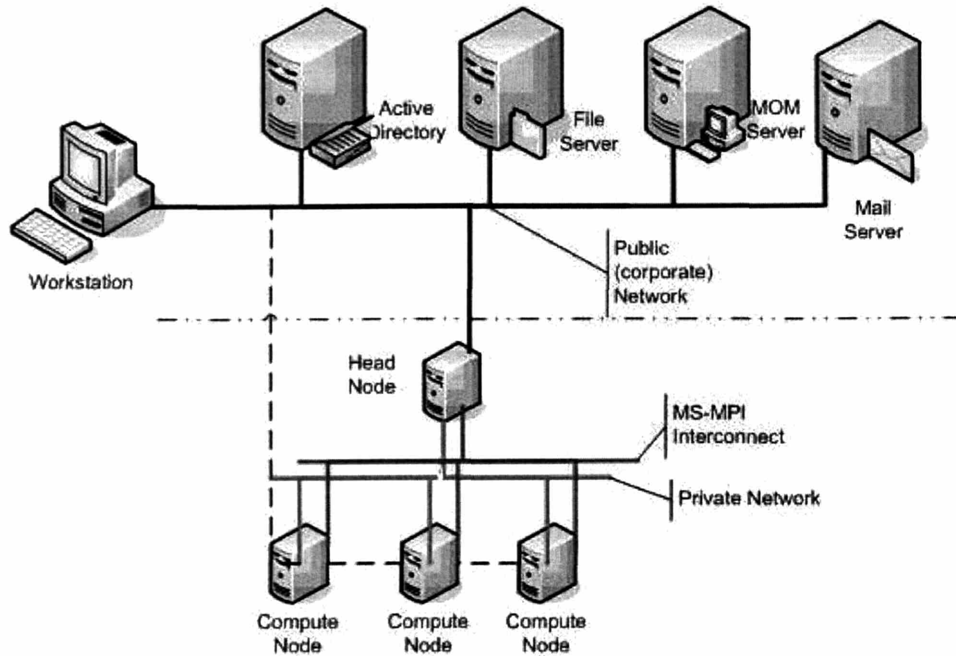


Figure 2-2: Microsoft HPC network (from Microsoft).

[33].

Microsoft Corporation has recently launched its *High Performance Computing* (HPC) project. After some investigations, we found that it is actually aimed at porting MPI to the Windows platform [26]. By doing this, the HPC project at Microsoft actually combines MPI and the Windows services such as file sharing and index services. The difference between Microsoft's implementation and the traditional MPI is that Microsoft HPC follows a master-slave topology with a head computer managing the computing nodes.

SETI@Home, distributed.net and MPI are just three examples of the old style distributed system. The common characteristics (or limitations) of these early systems are:

1. Many of them do not support node-to-node communication.

2. Some of them are *ad-hoc*, not for the general purposes. For these systems, distributed functionality has to be written from the scratch for each application (e.g. SETI@Home).

3. Old systems were built on top of early technologies that may not be attractive for today's ordinary users and application developers.

There has been much work done by different groups of people more recently. These projects embrace new technologies such as new programming languages, XML and advanced Remote Procedure Call (RPC) systems. They are discussed in the following section.

2.2.2 Recent Efforts toward Modern Internet Resource-Sharing

We have investgated a fairly comprehensive list of projects that aim to build modern distributed resource sharing systems. The technologies and designs of these systems are so diverse that we are not able to elaborate them one by one. In this section we provide a partial list as an introduction to these projects. The quoted comments are direct copies from their respective Web sites. We have benefited from various aspects of the projects listed below. For an exhaustive list of these projects, please refer to the distributedcomputing.info Web site [2].

1. Bayanihan. An academic work called "volunteer computing" was carried out at MIT [31, 30]. The approach was based on the use of Java Applets.[3] Java code was loaded into web browser's Java Virtual Machine (JVM), and subsequent computational instructions were received from the web server. The implementation was not complete, though, probably due to the lack of advanced software tools and network communication technologies. Anyway, this work provides some useful information about topics such as reliability under a hostile environment where malicious volunteers exist [31] and eager scheduling for embarrassingly parallel applications. We will discuss some of these topics later. The Bayanihan project is no longer maintained.

[2] More information is available at http://distributedcomputing.info/devel.html. The Bayanihan project is not listed there.

[3] Java Applet can run in Web browsers. It is a small piece of code downloadable from the Web site and executable on the local machine.

2. Globus Toolkit. "The globus Toolkit is an open source software toolkit used for building Grid systems and applications." "It is a fundamental enabling technology for the Grid, letting people share computing power, databases and other tools securely online across corporate, institutional, and geographic boundaries without sacrificing local autonomy."

3. Gridbus. "The Grid Computing and Distributed Systems (GRIDS) Laboratory at the University of Melbourne is actively engaged in the design and development of next-generation parallel and distributed computing systems and applications. The Lab's flagship "Open Source" project, called the Gridbus Project, is developing technology that enables GRID computing and BUSiness. The Gridbus project team is developing cluster and grid technologies (middleware, tools, and applications) that deliver end-to-end quality of services depending on user requirements. They include Economic Grid Scheduler, Cluster Scheduler (Libra), Grid modeling and simulation (GridSim), Data Grid broker, GridBank, and GUI tools for workflow management and composition of distributed applications from (legacy) software components.

4. N1 Grid Engine. Sun Microsystems' N1 Grid Engine finds idle resources on a LAN of Sun Solaris systems and uses them for your distributed application. N1 Grid Engine is available for Solaris, Linux, Mac OSX, AIX, HP/UX, Irix, and Windows Linux.

5. Alchemi. Alchemi is a .Net Grid Computing Framework. It is "a grid computing framework for Windows with the primary goal of being easy to use". It provides a) a programming environment to develop grid applications and b) the runtime machinery to construct grids and run grid applications. It is being developed as part of the Gridbus Project but unfortunately not compatible with other components of Gridbus system.

The long list of these projects does tell something. Driven largely by the needs of business-to-business computing, new communication standards have evolved. XML

based technologies, such as SOAP, have been adopted by a large number of software companies, including IBM, Microsoft and Sun. Also, the Microsoft Windows platform has become the major operating system for desktop computers. Given the rapid emergence of advanced tools and packages on Windows, such as the .Net Framework and their integrated development environment, research on developing a modern distributed computing environment for Windows is indeed worthwhile. Ideas aiming to embrace recent technologies and be applicable on Windows platform thrived, especially between 1996 and 2003.

These projects demonstrate great creativity and intelligence. It is unfortunate that none of these projects was proved to be a significant one of the current most influential technologies. Most of the small projects are no longer maintained. This is probably due to the limited functionality, limited financial support or less attractive design. For example, those having no support for node-to-node communication, such as Bayanihan, found it difficult to expand their user base.

The current commercial or other continuously supported projects, such as Globus, Gridbus and Sun N1, have not established their significance in any industry, though the quality of them is indisputable. Some projects also published their methodologies as standards. In particular, *Grid Computing* has been acknowledged as the most promising of future Internet resource-sharing methodologies. The concept of the grid, borrowed from the "electricity grid", is referred to as the sharing of computation power and storage resources [39]. The interfaces for individual grids are not standardized. The communication protocols between grids are also vendor-specific. In 2002, the Open Grid Service Architecture [1] was proposed to define stateful grid services with lifetime management and other semantic descriptions about grid resources. The main idea of stateful grid services is to use Web services to define the grid resources associated with state information. The current Web services framework available for grid services is the WS-Resource Framework [15]. Other than the tools provided with these systems, third-party support is seldom seen for these "open standards."

In my view, the limited usage of the current products and the reluctant compliance to their standards are due to their lack of accessibility and programmability for

inexperienced computer users and ordinary programmers. For example, the OGSA standard comprises a fairly demanding list of functionality. The implementation, Globus Toolkit, is thus unavoidably complicated. This makes its installation and application programming difficult. Even for just understanding the full system, the learning curve is steep. Also, backward-compatibility is often overlooked for those software packages that have limited number of users. Although this is actually an optimal strategy to shorten the development cycle, the application developers suffer from the inconsistency. More importantly, greatly the complicated design of these toolkits makes some of these projects lag behind schedule. For example, while the Globus Toolkit version 2 had not firmed its standing in the market; its underlying technologies, which were mostly based on the C programming language and direct socket connections, were obsolete. The far more stable version 3 took some time to introduce XML and Web services into the implementation.

One project meriting discussion in greater detail is the Bayanihan project. It proposed a Web based distributed computing methodology and a runtime system that had the capability of running distributed code based on Java. Although it is actually more a proof-of-concept prototype system, some parts of the author's study are very helpful, especially the reliable scheduling for master-slave applications under the attack of malicious volunteers. His work also provided a pool of application scenarios. It is an important reference for this thesis.

In general, the old-fashioned and modern distributed systems still lack a lot of desirable features. We have analyzed the experiences and a few lessons from these previous works. We then proceed to propose our own idea.

2.3 The Realm Framework

We propose the *Realm* as a new idea for sharing computer resources across the Internet. The Realm Framework seeks to meet the requirements of the majority of Internet users and distributed application programmers by using the most recent technologies. The computer resources it is expected to support include processing power (typical

parallel computing task), storage and mobile devices. Due to the lack of the necessary software tools to write and test code on mobile devices such as the PDAs and cell phones[4], mobile device support is not in the implementation described in this thesis. We only glimpse the idea of distributed devices in this section.

In this section we demonstrate the Realm idea with a series of potential use cases. With this general outlook, we then derive the feature set that the Realm Framework most have.

2.3.1 Potential Application Scenerios

All scenarios except the use of distributed mobile devices have their corresponding sample applications detailed in the later part of this thesis. Please refer to Chapter 4 through 7. As we have mentioned above, the distributed mobile device idea lacks the support of development tools and thus was not implemented. For the same reason, the distributed mobile device scenario is very general instead of being realistically detailed.

Master-Slave: The Computing Style for Embarrassingly Parallel Problems

We usually call a problem "embarrassingly parallel" if the work-flow of solving the problem can be partitioned into completely independent parts. For example, when we use the naive brute-force solution to crack an encrypted password, the trial-and-error steps do not rely on any information other than the pass code to be tried. A simple master-slave topology is commonly employed in practice (Figure 2-3).

The master dispatches jobs to the slaves one by one and collects results from the slaves. In spite of the fact that only a small portion of the real world problems can be solved in a truly embarrassingly-parallel approach, this simplest form is an ideal start-point of studying various aspects of a distributed/parallel computing system. This is because the independence of the sub-parts makes adaptive parallelism and fault-tolerance easier to achieve than in other applications. In particular, since the

[4]This is mainly due to the device's capability but not the mobile technology in general.

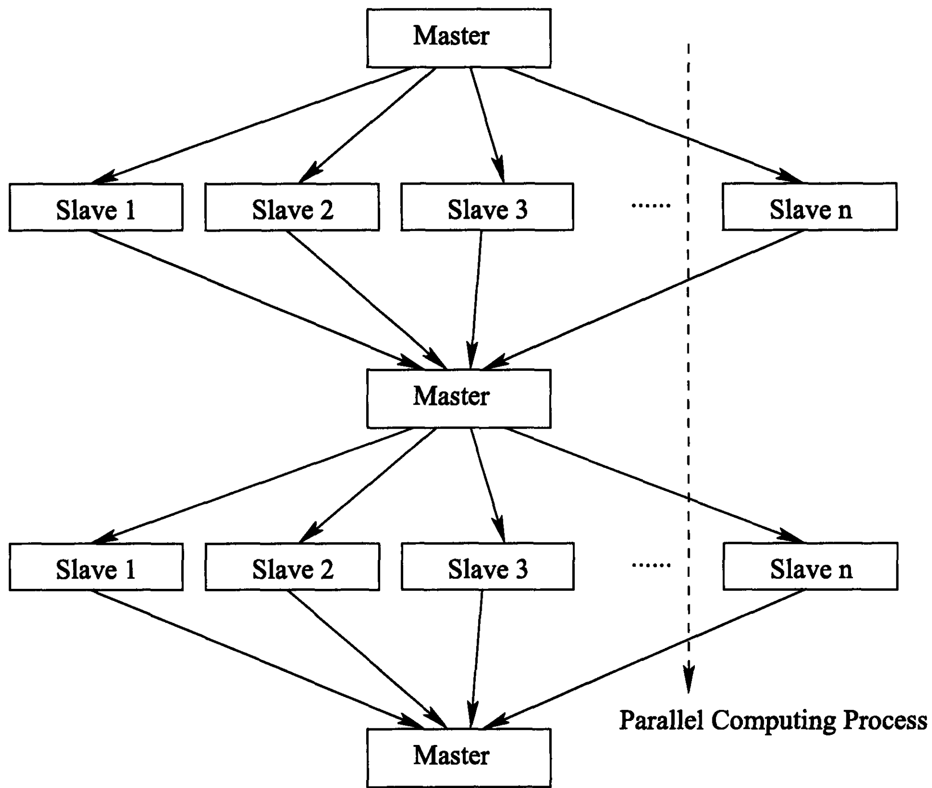


Figure 2-3: The simple Master-Slave style parallel computing topology.

individual processes need not be aware of each other, the master can easily re-schedule the sub-jobs in case of failures or malicious attacks. The early projects SETI@Home and distributed.net, demonstrated the typical model for solving embarrassingly parallel problems.

Communication-intensive Parallel Computing: The Computing Style for a Broader Range of Problems

While the master-slave model covers a number of applications, these represent only a fraction of the variety of parallel applications. In particular, most parallel applications today, such as those written with MPI, assume a message-passing model. In this case, the computing nodes that have been assigned to a distributed job are aware of the existence of each other. They can also locate the other nodes and send or receive data to and from the others. Typical engineering problems, such as fluid modeling, require this kind of distributed computing model because these problems are usually not able to be partitioned into completely disjoint sub-jobs. The message-passing-based distributed computing system typically needs much more maintenance and programming work to be usable and stable. The Realm Framework assimilates MPI's model (Figure 2-4) when it is used to solve complicated problems for which the simpler master-slave model is not applicable. Due to the added complication of the message-passing, the Realm system, like MPI, provides little support for situations where nodes leave or join the job at unexpected times. It does not support re-scheduling and fault-tolerance if running in the message-passing style. For these reasons, this computing model is best working over a group of closely clustered machines or within a private network.

The Realm idea is different from the previous systems in a few ways. First of all, people do not have to wait for the completion of the application program. They can access the intermediate results even while the program is still running, which is difficult to achieve with MPI. Also, the Realm has direct support for developing a message-passing-based application. Since Grid Computing solutions all aim at general purpose resource management, they still need a runtime system to do parallel

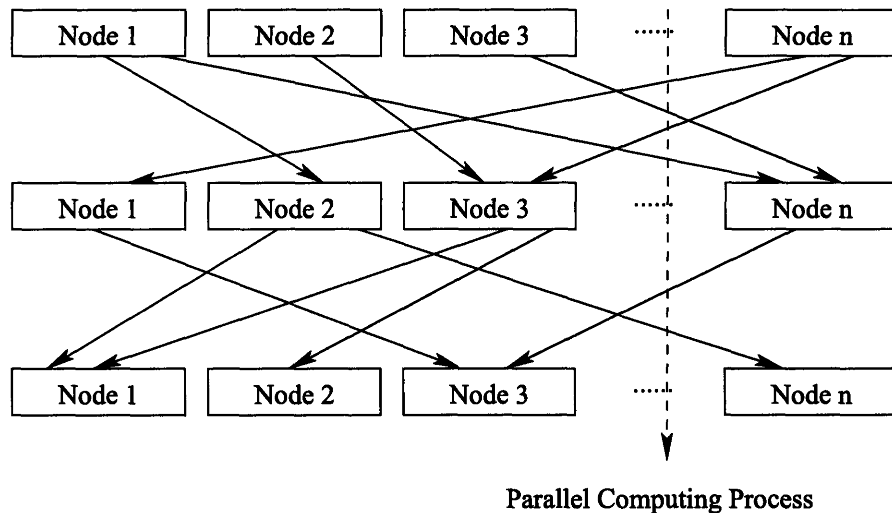


Figure 2-4: The MPI-style parallel computing topology.

computing. A sample of this combination is MPICH-G2 [22]. Programming and management of this system, however, tends to be difficult because MPI and Grid Computing are two distinct systems. There are compatibility issues to be solved, which in turn restrict the stability and usability of the MPICH-G2 system.

Universal Object Storage

Data storage is another computer resource that can be shared across the Internet. Many protocols and systems have been developed to achieve reliable and efficient *file-sharing*. For example, using Microsoft Windows platforms, the computers talk in the *SMB* [5] protocol [25] to read or write files on the other computers; the systems with MPI usually apply NFS [33] to share a common hard drive space so that computing processes running on different machines have consensus for data input and output. In Globus' grid computing solution, gridftp is applied so that a huge data file can be distributed and saved over a group of grid points [1]. The common characteristic of all of them is that they are file-based. The modern Object Oriented (OOP) programming languages, such as Java and C#, deem all data to be objects. Extracting and identifying objects safely from files is tedious work and not consistent with the the

[5]SMB stands for Server Message Block. Microsoft uses the so called NT LM 0.12 variant on its Windows Operating System.

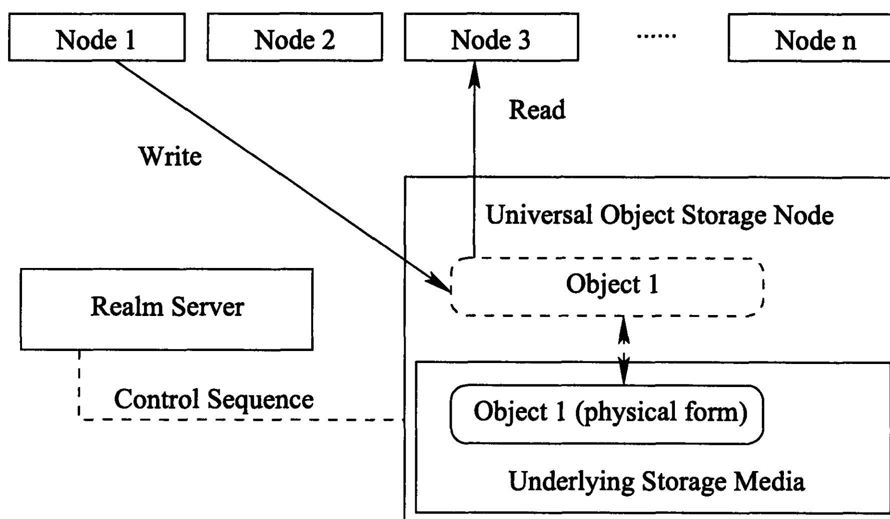


Figure 2-5: The interaction between a Universal Object Storage node with other nodes.

goal of OOP. Also, some of the previous systems are not adaptive to the modification of the network configuration. For example, on Windows, we need to search the remote file again if the computer name or the share point has been changed. This needs considerable administrative work and is not flexible.

The Realm concept includes support for distributing data objects, not files, with a fixed retrievable Internet name. From the application programmer's point of view, they can save and retrieve data objects directly with a string which does not depend on the location and the backend storage media. To achieve this, a *Universal Object Storage* process runs on the machine that offers the shared storage as shown in Figure 2-5. The backend storage media can be virtually any readable media with optional writing capability, accessible by the machine running the shared storage. The storage process acts like a broker between the data user and the backend storage media. To be "universal", the accessing interface is common across all nodes.

One disadvantage of the Universal Object Storage model is the capacity of data it can support. Since the object needs to be instantiated and the object is passed as a whole, for huge objects such as an extremely long data array, this burdens the system memory and the network. A work-around is to avoid putting everything in a container object. Instead, we can divide them into fine-grained objects. This works

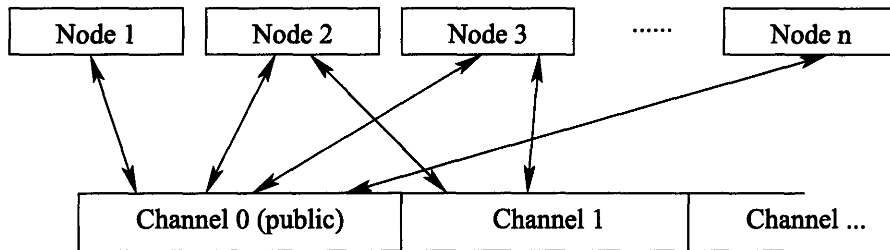


Figure 2-6: The channeled communication style.

well for the sample applications we have written with the Realm Framework.

Working as an Instant Messenger

The capability of message-passing, node-coordination and address-lookup makes the Realm possible middleware for an Instant Messaging system. By providing a number of *channels* with pre-defined security configurations, the Instant Messaging mode satisfies the requirements of most Instant Messenger functions (Figure 2-6). For example, in a group meeting, the general messages pass through channel 0 which is a public channel. Channel 0 reads all messages passed to it and broadcasts them to all the meeting participants. In case person A needs to talk to person B privately when the meeting is going on, person A acquires a channel, say channel three, and configures it as a private one with only two participants, A and B. The Realm then notifies B to initiate the talk. In Figure 2-6, node 2 and 3 are communicating through the private channel 1. The channels support file and image transfer by wrapping them into objects.

The model can be used not only for person-to-person communication, but also for business transactions. A simple application for the logistics industry is demonstrated in Chapter 7.

The Prospect of the Mobile Device Network

What if the Realm concept generally introduced above is applicable to mobile devices? This will bring us to a new way of consuming Internet resources. PDAs or cell phones can be used to view the real-time progress of a parallel job; shipping and delivery

information can be directly accessible from the bar code reader which is in the Instant Messaging network of a supply chain. All of these are done by a single system. The productivity of individuals or corporations can be greatly improved.

2.3.2 Features to be Implemented

From the typical scenarios listed above, we can derive a set of features that need to be implemented to realize the Realm.

1. The Realm Framework must be easy for programmers and end users to use.

2. The Realm Framework maintains a runtime system that can manage and execute code written for it.

3. The Realm Framework maintains a messaging system that is able to reach any participants within the Realm, and preferably has a scheme to communicate with other Realms.

4. The Realm Framework maintains a location system that records the address of each node of the Realm, and offers a convenient method to reach them.

5. The Realm Framework is sufficiently versatile to support various use-cases mentioned above.

6. The Realm Framework should have developer's tools, particularly a debugger designed to support distributed applications, to facilitate application programming.

7. The Realm Framework must be secure and robust, and ready to be exposed to the public network.

8. The Realm runtime system needs to be reasonably reliable in terms of network or hardware failure.

9. The Realm Framework incorporates the most recent technology, and is incrementally implemented with short development cycles to make sure its up-to-date.

2.4 Research Topics and Challenges

Although distributed computing is not a new research area and there have been many previous accounts and prototypes available, realizing the features mentioned above and implementing a functional Realm system with the most recent technologies involves several challenges. These can be classified generally into accessibility, programmability, reliability and efficiency. In this section we discuss all these issues that have to be faced when developing the Realm Framework.

2.4.1 Accessibility

We define the accessibility as how easy we can use the system. To an end-user, the usability of a software system is a significant issue. This is especially true for casual computer users. For old systems like MPI, this problem is unimportant because their targeted users are limited to those knowledgeable about the Unix system. Also, MPI systems are usually dedicated machines co-located together without an ordinary computer user sitting in from of them, MPI-like system designers tend not to worry about the ease-of-use problem. However, the Realm idea allows more people to join the share-pool. As I have mentioned before, the majority of computers are running Microsoft Windows, and most computer users are not professionally trained in system setup and administration. The Realm system must be intelligent enough to do as much work as it can by itself, but offer detailed configuration options when necessary as well. The Realm system should support at a minimum the Microsoft Windows platform, and possibly others as well. This was difficult previously, but with the support of virtual machines, this can be done with a careful implementation. Finally, for end-users, a Graphical User Interface (GUI) is always welcome. Although GUI development is usually less diffcult than system development, this inevitably introduces

added complexity in implementing the Realm system.

2.4.2 Programmability

Programmability under our context means how well the distributed system can address developers with widely differing backgrounds and development goals. Or in another word, a well-programmable system should be versatile under different use cases. A distributed system is not interesting if it is not useful. The Realm idea is to provide middleware so that people can run distributed software on it and benefit from the resource-sharing capability offered by the Realm system across the network. As programming framework, the Realm system is not obliged to directly support the needs of the end-users. The applications, such as the parallel program to crack an encrypted code or a universal storage program that supports the SQL server as its backend storage media, are actually written by the application developers. They decide how to make use of the Realm Framework. How conveniently they can use the Realm Framework to write application software is critical to the future of the Realm concept.

The Realm Framework offers multiple layers of programming interface to fit the needs of the programmers with a diverse background. For example, researchers in engineering may feel comfortable with a parallel programming interface that has encapsulated parallel computing techniques such as the ghost area, time step synchronization or even automatic domain partitioning. They do not like and they do not need to know what is going on at the backend. A higher-level abstraction close to the real engineering problems is what they want. On the other hand, some more professional programmers may feel that they could use the Realm Framework as a skeleton to build a mobile-device network. Since the mobile-device network is not yet supported in Realm system, they can pick a lower development layer from the Realm system to build an extension for their specific mobile devices.

2.4.3 Reliability

Because of its open nature, the Realm Framework is more prone to faults or attacks than other forms of distributed computing systems. The Realm system, as a reliable software package, should be both fault tolerant and robust with respect to any possible malicious behavior.

Fault tolerance includes dealing with the situations such as node crashing or leaving, and malfunctioning. The former problem is usually called *stop failure* [24]. This is the simplest form of fault and has been studied thoroughly. The latter form represents both accidental malfunctioning and intentional malfunctioning where the node intentionally submits erroneous results. It is also called *Byzantine failure* [24] in some of the literature. The Byzantine failure is more difficult to handle. Early systems for simple embarrassingly parallel problems are believed to support fault tolerance for stop failure. The Bayanihan project also suggested a simple scheme to examine the ability of a computing node to guard against Byzantine failure. More sophisticated systems vary in supporting fault tolerance because it is extremely challenging to develop a full-functioned system that is always fault tolerant. For example, MPI paid little attention to fault tolerance. In Realm system development, we have decided to consider fault tolerance only in the case of embarrassingly parallel problems. More general forms of fault tolerance are beyond the scope of this thesis.

To protect the Realm system against attack, two things need to be considered. The first is runtime security. When passing executables around the network, there should be a guarantee that running the code is safe. Restrictions on file system access, network access and so on are often imposed. Some virtual machine systems allow us to run client code in a restricted *sandbox*. Both Java and .Net Framework have this functionality. In Java, the Java Applet always runs in the Java sandbox [35]. The Bayanihan project has already taken advantage of this [31]. For .Net, the programmers can specify security restrictions to code from the Internet [6]. We apply this technology when we develop the Realm runtime system. The other issue to be considered is network security. A system exposed to the wild Internet is automatically

[6] Related topics are available at MSDN.

an open target for remote attack. Without considering security, reliability is hard to guarantee. Encrypting connections in the Internet environment has become a common requirement for most of distributed systems, especially for commercial usage. Open Internet connections can leak critical information to a third party. An authentication scheme is also necessary to limit the service to authorized users. An unguarded universal storage can quickly break down if someone intentionally sends huge data objects to it. There are many important previous works we can refer to, such as Kermit [20], the secure HTTP scheme, and a series of encryption algorithms. It is still challenging to pick and weld these technologies into one integrated package. In Chapter 3, we will discuss the .Net Remoting secure communication method used in the Realm system.

2.4.4 Efficiency

Another challenge in developing a distributed system is how to make it efficient. Efficiency can further be divided into two types, runtime efficiency and programming efficiency. By choosing an efficient programming language such as C# or Java and carefully considering the programmability issues mentioned above, programming efficiency is easily achieved. Runtime efficiency has multiple facets. The executables themselves may or may not be efficient, depending on the compiler and underlying runtime system we use; using different communication protocols will also affect performance; design and implementation in general, including synchronization, load balancing and so on, also affect the efficiency. There are trade-offs among these aspects of efficiency which also add complications to the research.

2.5 Context and Conclusion

We aim to build a new computer resource-sharing framework that satisfies the needs of today's users and incorporates the most recent technologies. In this chapter, we presented the potential use-cases for a new resource-sharing system within the context of current technology and modern users. We then shortlisted a set of features that

are desirable in this context. When we develop such as system, there are indeed many challenges and interesting research topics we have to address. We will demonstrate how we solve them in the next chapters.

Chapter 3

The Realm Runtime System

We have reviewed the previous work in Chapter 2 and proposed a new distributed system, called the *Realm*, which has many features favored by today's end-users and application programmers. In this chapter, we present a Microsoft .Net based implementation of the Realm concept. We begin by discussing the disadvantages of some previous implementations of distributed systems and then explain the advantages of using XML-based technologies and the Common Language Runtime to implement the Realm system. We then present the design of the system and demonstrate how the Realm system achieves: 1. better accessability than earlier systems to help more people use it, 2. better programmability than conventional parallel computing system for different levels of programmers, 3. better reliability in the context of today's Internet environment, and 4. quality runtime efficiency without sacrificing functionality.

3.1 The Development Platform: .Net Framework and C#

Many people in the parallel computing world like to talk about *extreme performance.* The software program running under a parallel computing framework has to be very fast in execution. Under this context, fully-compiled binary files are commonly used as the executables. Programming languages that can be compiled in this way are

often chosen for this purpose, as is the case of MPI. In MPI, C/C++ are the default programming languages. Other programing languages available for MPI, such as Java, are not seen in a significant number of cases, though the Java binding for MPI does work [4]. The available programming tools are then limited to the C libraries and C-related programs that have only a small function set. Moreover, they are developed by a number of different companies or organizations and thus suffer the problem of incompatibility.

We use the Microsoft .Net Framework and the C# programming language as the underlying platform to develop the Realm system. The .Net Framework offers numerous components for us to develop the Realm system. The *Common Language Runtime* (CLR) [1] is similar to Java bytecode and it offers virtual machine many desirable features, especially platform-independence.

3.1.1 Limitations of the Traditional Systems

Using traditional programming languages such as C and C++ suffers from the following limitations.

1. The fully-compiled, machine-code based binary is not suitable for large scale Internet sharing. Imagine a hybrid network consisting of some Microsoft Windows machines, some Apple Mac machines and some machines running Linux. The application programmer would prepare three executables for a single job so that she or he can take full advantage of the computing network. This not only burdens the application programmer, but also complicates the management of the executables and computing nodes.

2. The C programming language is not Object-Oriented. Without the features of OOP, software programs written in C are not easy to re-use and extend.

3. The native machine-code executables have some disadvantages compared to virtual machine based executables. For example, garbage collection and correct

[1] For CLR related standards, check http://msdn.microsoft.com/netframework/ecma/.

memory usage are usually the responsibility of the programmer in machine cold executables. The development cycle is often longer than in those projects using Java or C#. The software quality needs to be checked with extreme caution.

4. Without integrated, advanced and high-performance programming packages, those features today's users mostly favor will take an unreasonably large amount of time to implement. For example, it would be expensive and redundant to develop another Web service layer from scratch. The Globus' Grid Computing implementation collects many third party software packages to take advantage of recent technologies [2]. This implementation is nevertheless troublesome to keep updated when any one of its component packages releases a new version. Moreover, because those third party packages follow a different programming style or even use different programming languages, putting them together is far more difficult than using them individually.

3.1.2 Advantages of Using the .Net Framework and C#

1. The .Net Framework supports the CLR, which is an advanced and platform-independent runtime format. C# source code can be compiled into CLR byte-code and distributed to other machines for execution without worrying about the host operating system and hardware architecture. This enables us to expand available resources much more broadly than in older systems. Besides, the CLR allows some advanced functions such as versioning. These ease the management of executables.

2. The C# programming language is Object-Oriented and comes with many very useful programming tools for code re-usability.

3. In the .Net Framework, the CLR runs in a virtual machine rather than directly accessing memory. The virtual machine offers efficient garbage collectors and is

[2]For example, the most recent version of Globus Toolkit 4 needs Java, Ant, C compiler, C++ compiler, tar, sed, zlib, gmake, perl, sudo, JDBC compliant database, gpt, IODBC, Tomcat and gLite pre-installed to support Web services enabled Grid computing.

immune to memory leakage in most situations [3]. Programming within the .Net Framework is fast, and it is easy to write high-quality code.

4. The .Net Framework offers many libraries and components ready to be used by the application programs. Without any other software package, the Realm is still able to provide a rich set of features.

5. The C# and CLR are open standards. More and more software developers are becoming C# programmers and distributing CLR-based executables [4]. C# is also easy to learn. For these reasons, we do not have to worry about a shortage of programmers.

3.2 The Realm Distributed Resource Sharing Runtime System

3.2.1 Goals of Design

To demonstrate the benefits of the new concept of distributed computing, and to explore the different aspects of this idea in general, we have developed the Realm runtime system with Microsoft .Net and the C# programming language.

We have the following goals in designing this system:

1. To maximize the accessibility and programability so that the system will be used by a large number of end-users and application programmers.

2. To meet the high standards of Internet security so that the system can be immediately deployed on the Internet.

3. To be as efficient as possible in terms of runtime and programming time.

4. To be versatile so that researchers can take advantage of it in many use-cases.

[3] C#, as well as Java, is not perfectly free from memory leakage, though it supplies garbage collecting. However, in most cases, we will not encounter the memory problem.

[4] They are also called "managed code".

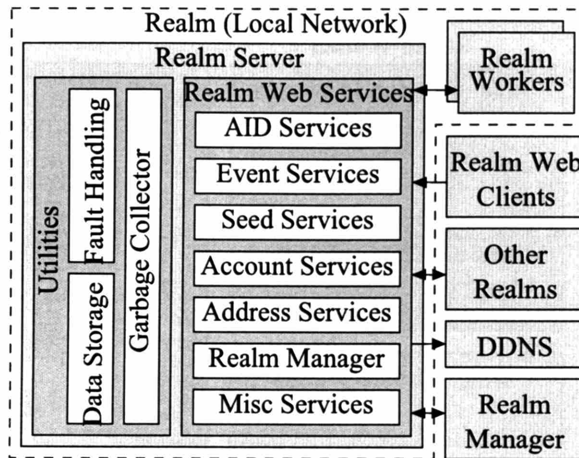


Figure 3-1: Components in the Realm Framework.

In this section, we examine the general design of this software framework and discuss the approach we take to achieve these goals.

3.2.2 System Design

The basic components of the Realm system are showed in Figure 3-1. This is a master-slave architecture in which we call the master computer a *Realm Server* and the slave computer a *Realm Worker.* Noted that this "master-slave" topology has no relationship with the embarrassingly parallel model we will discuss later. In the context of the embarrassingly parallel model, the master computer is not the Realm Server. Instead, the master computer in that situation refers to one head Worker and the slaves are all other Realm Workers assigned to the job. Actually, we could design in a way that we assign the Realm Server as the master node in the case of embarrassingly parallel problems. We do not do this because the Realm Server would be heavily burdened with two tasks in this strategy. Computation should be passed to the Realm Workers as much as possible. The computing is done through a number of *Worker processes.* Workers (Figure 3-2), the computers that hold the Worker processes, are distributed either within the local area network or over the Internet.

Programs running as Worker processes are coded in the CLR runtime. The application program is normally a CLR dynamic loaded library (DLL) with entries

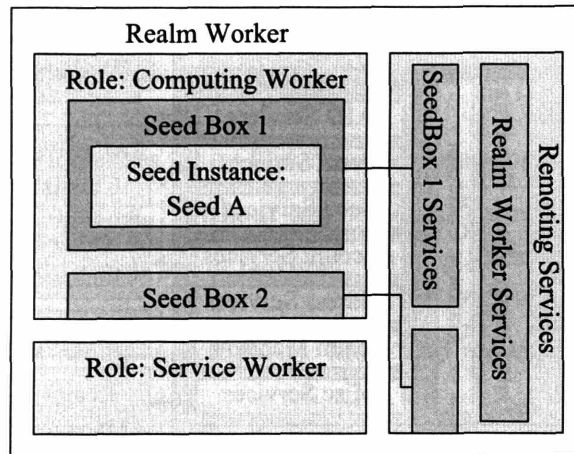


Figure 3-2: Structure of a Realm worker.

accessible by the Realm system. We call the DLL a *Seed*, a vivid name suggesting its duty–a small mobile unit that can be passed to another system and do its predefined job. The Realm Framework stores the DLLs in a component called the *Seed Pool*. Abstract Seed types, as the skeletons to develop Realm applications, are categorized based on their usage. Some often used Seed types are: *MultiStore* for the message-passing style parallel computing, *Star* for embarrassingly parallel programs, *Storage* for Universal Object Storage, and *Messenger* for Messenger-like programs. Typically, the application programmers of the Realm application populate functions based on these abstract classes to do their specific jobs, such as scientific simulation or business transaction.

It is time to introduce the *Net Application Domain* (NAD). Similar to MPI's "Communication Group", each *Realm job*, which is executed by a group of Workers, is assigned a NAD so that the job can be easily identified and located. The NAD facilitates identification and communication of Worker processes in a Realm application. One difference between a NAD and MPI's Communication Group is that the NAD is not just for communication. It provides a shared memory block for these worker processes which can be accessed from the Internet. The shared memory block, so called *Global*, is critical for Realm-based applications. If Realm users want to access information about a Realm application or to control the application, they can do this

by reading or writeing Globals. Any serializable objects in C# can be stored as Globals. Different from the Universal Object Storage discussed later in this chapter, the Realm Globals reside directly on the Realm Server and are lost if the Realm Server is shutdown or restarted. Also, to make sure the Realm Server is not brought down by a large volume of data, each Realm job has an upper limit on the Global storage size.

3.2.3 Process Management

The Worker processes are managed by the Realm Server. On the Worker machine, the current computer operator has little control of the processes other than defining the maximum number of processes, defining the Worker's role (Computing, Storage or Messenger), and the ability to shutdown the Realm Worker system. The process management scheme proposed in this design has the following features:

1. Each individual Realm job is identified by a NAD ID, which is unique across the network. Each NAD ID is separated into two parts. One is the Realm's Internet ID (supported by the Dynamic DNS system discussed in Chapter 8) and the other is a unique ID for the job inside the Realm. So for each Realm job, there is a universal name that can be used to locate it.

2. An individual computing process running on a Worker machine is identified by an integer starting from 0. This is similar to MPI. This abstraction is necessary because using IP address or other means complicates the programming, and this is a familiar way that most MPI programmers understand.

3. Computing threads inside the "SandBoxes" (see Figure 3-2 on page 46). The idea of a SandBox is borrowed from Java Applet [35]. Introducing the concept of SandBox not only makes the managing of individual threads much easier; it also provides a security boundary between the system and Realm executables.

4. A SandBox listens to commands through a .Net Remoting Channel on a specific port. The .Net Remoting is an RPC sub-system that offers an excellent

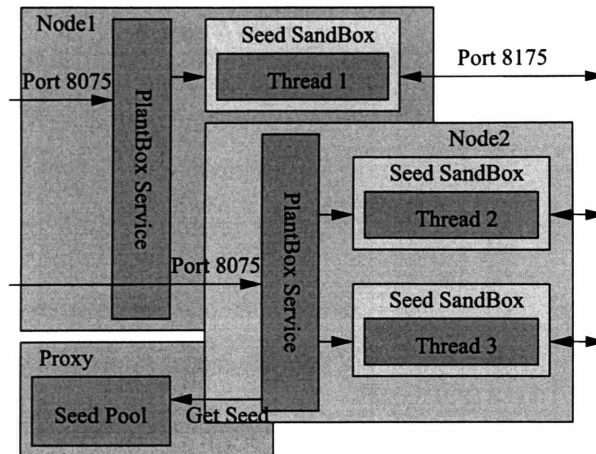


Figure 3-3: Creation of the Worker processes.

abstraction for network communication. The underling connection can be TCP, HTTP or even a customized connection protocol. We will discuss how to use .Net Remoting to build a flexible and secure communication scheme in this chapter.

5. On port 8075 of each Worker machine, there is a .Net Remoting service *PlantBox* used to listen for Realm commands. Commands issued to this port are usually for the creation and termination of Worker processes.

Figure 3-3 describe how computing threads are created. Upon receiving a job request from the Realm Server through port 8075, the Worker machine fetches the proper DLL (Seed) from the corresponding service of the Seed Pool on the Realm Server. SandBoxes are created, and free network ports are opened. The SandBoxes publish new Remoting services through these ports. The SandBoxes report the new port numbers to the Realm Server so that the Realm Server and other Realm Workers are able to contact the SandBoxes. The Worker processes are then created inside the SandBoxes, following the security policies imposed by the SandBoxes.

When it receives any commands or data, a SandBox passes the information to the Worker process. This is done by calling corresponding methods of the Seed object. These methods should be implemented by the application programmers.

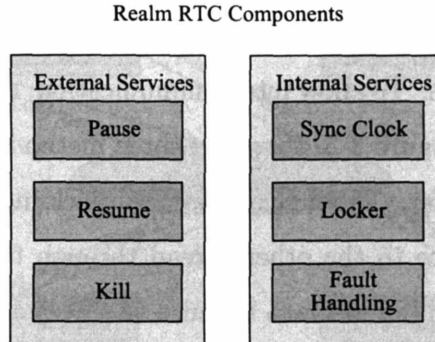


Figure 3-4: Runtime control components.

To allow users to issue instructions to a Realm job, there must be an interface between the Worker process and the user. The Remoting service exported by the SandBox is not enough because it is used for an individual computing threads but not all the Worker processes of a Realm application. To achieve runtime control, the Realm Server is a reasonable entity for hosting the interface. In fact, as shown in Figure 3-4, the Realm Server has two sets of process control services. The external services are exposed through an HTTP Remoting channel as Web services. This allows users to control the Realm application from the Internet using any tools provided they follow the standard of Web services. The internal services are exposed as Remoting services, too, but are only used inside the Realm. Note that the boundary between the Realm and the external world is not necessarily the same as the boundary between a Local Area Network (LAN) and the Internet. For a message-passing style application, a Realm is indeed running inside a LAN for the sake of performance. However, other applications can take advantage of the security communication layers offered by the Realm Framework to form a safe virtual private network on the Internet.

3.2.4 Intercommunication between the Worker Processes

Many parallel computing algorithms are not embarrassingly parallel, which means they require communication among Worker processes. Even for the problems that can

be partitioned into independent sub-jobs, a communication system is still needed in our design because the master node is not placed on the Realm Server, but resides on one of the Worker processes. To allow intercommunication, a powerful communication scheme is developed. In Figure 3-5, there are three methods for a Worker process to send data to another Worker. The first method is "quick messaging". Small amounts of data can be sent at once to the other thread through the Realm Server. This is not suitable for a large volume of data because the Realm Server tends to be much busier than the Worker machines in terms of network connections. This motivates the need for a second communication method, the "big object messaging". When sending data, the sender actually only sends a reference message to the other Worker via the Realm Server, telling the other Worker process that there are data ready to be picked up. This avoids a network traffic jam on the Realm Server but may introduce a small latency. This second method allows the receiver to decide whether and when to pickup the data. This may reduce the network traffic. The last communication method is to connect directly. This is allowed in the framework but is not recommended because Worker machines are much more vulnerable to failures than the Realm Server because they run jobs and the jobs are mostly computationally intense. If a Worker process fails, other Workers sending messages to it may stall or even crash. The Realm Server, however, has a mechanism to detect failure by pinging the service of each Worker machine periodically. In case of failures, the other Worker processes will at least get this failure notification before or after sending messages to the failed one.

The first two methods only involve short messages to and from the Realm Server, which is very similar to the event system used in many operating systems. For this reason, this scheme was named the "Event Sub-system".

Communication reliability is guaranteed by a reference number attached to each message, and a re-sending loop on the sender side. On the receiver side, special handling of received messages ensures the uniqueness of a single message as well as the order of the messages.

In this section we have discussed the general structure of the Realm Framework and its components. We also demonstrated how these components work together.

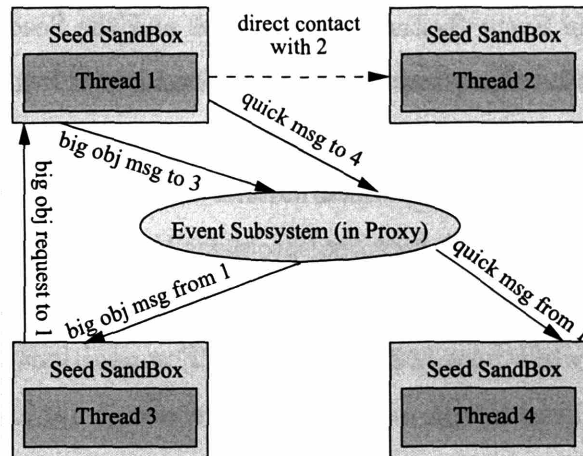


Figure 3-5: Communication scheme.

As has been mentioned in the last chapter, we have faced challenges in considering the accessibility, programmability, reliability and efficiency of the Realm. In the next sections we will show how we solve these problems.

3.3 Accessibility

The Realm system is designed to be easy to use for end-users, who may even be casual computer users with very limited knowledge about any computer systems. The application programmers, although supposed to be more knowledgeable than ordinary users, commonly favor great usability so that they can develop applications more productively. In this section we demonstrate the basic procedures to run a Realm system as well as write code for it.

3.3.1 Realm Setup

Setting up a Realm has two steps–the Realm Server setup and the Realm Worker setup. Both of these have a default setup scheme for the majority of users. This "impatient" scheme does not require any modification of the configuration files, and starts the system with just a click on the executable files. The Realm Server is a Windows console application (RealmServer.exe). When launched, the Realm Server

program activates the Event System, Web services and the Seed Pool in a chain. It also waits for network connections on ports 8080 and 8081. In all cases, the 8080 port, which handles requests from the outside the Realm, should be open to the Internet. Whether the port 8081, which handles requests from inside the Realm, is open to the public depends on the actual problems and the parallel computing style we are using. By default, the 8080 port supports HTTP connections with encryption and authentication, while port 8081 supports TCP connections with authentication only [5]. More possibilities for the communication channels are discussed later in this chapter. The RealmServer.exe needs to read a series of environmental variables predefined in the "env.bat" [6]batch file. These variables tells the Realm Server where the configuration files and executables are, and the location of the Dynamic DNS server, which is discussed later. The default settings do not have the Dynamic DNS configuration; in this case Internet users can only use IP addresses to locate the shared resource.

The Realm Worker is an application called WorkerI.exe on the Microsoft Windows platform, or a console program called WorkerC.exe on other platforms that support Mono [7]. The WorkerI.exe starts a GUI on the Worker computer, as shown in Figure 3-6. The end-users use this GUI to communicate with the Realm Server and control the behavior of the Worker processes. The GUI begins with a log-in form to join the Realm. The parameters, such as the role (computing, storage or messenger), the maximum number of processes and user's identification need to be filled in and passed to the Realm Server. After joining the Realm, the end-users can choose Realm applications from the list of Seeds. The execution of the Seeds is defined to be either *network* or *local* by the application developer when they wrote the program. In "network" execution, the Realm Server spreads the executables over the network, which is the typical mode for a parallel computing application. In "local" execution, the Worker machine starts the code, typically used for storage and messenger type

[5]In Chapter 5, we will show a comparison to explain why we use this setup by default.

[6]Or the C-Shell script "env.sh" under FreeBSD.

[7]Mono is an open source framework which is very similar to .Net Framework. It is available at http://www.mono-project.org.

applications.

The GUI also allows the end-users to get information about the running jobs. As we have mentioned before, the jobs are identified by a unique NAD ID. With this ID we can fetch the data objects stored as the Globals, check the number of Worker processes, know which Realm Workers are assigned to the job and control (pause, resume or stop) the job. One inconvenience of using the NAD ID is that the system-generated NAD IDs are long and often not human-readable. To facilitate people using the framework, the Realm system allows an alias to be assigned to the job. Although internally everything is bound to the NAD ID, the more user-friendly alias offers great flexibility. The WorkerI.exe GUI also provides some other useful functions like management of the local Worker processes and the ability to communicate with other end-users. Most of the functions in WorkerI.exe are supported in WorkerC.exe, too, in a command-line fashion with help from another program called SeedRun.exe for code execution over the network. Unlike MPI, the Realm Worker is added to the network with little or no configuration. This is more like the style proposed in the Bayanihan project, where the "volunteers" can visit the Web front-end of the server and load the executables at any time from anywhere.

3.3.2 Seed Programming

Microsoft Visual Studio comes with a convenient programming GUI for C# and can be used to develop Realm applications. It supports many useful features such as code auto-completion, syntax highlighting and an integrated environment for code writing, compiling and debugging. The only additional work is to put the Realm's development library *GridLib.dll* into the library reference list. Additonally, a Realm Seed template package has been developed for Visual Studio. With this package, the application programmers can create a skeleton project from the template without worrying about the library reference. For those who do not want to pay for Visual Studio, free alternatives are also available. The Visual Studio has a slimmed-down free version called "Visual Studio Express". Another development GUI is the open source

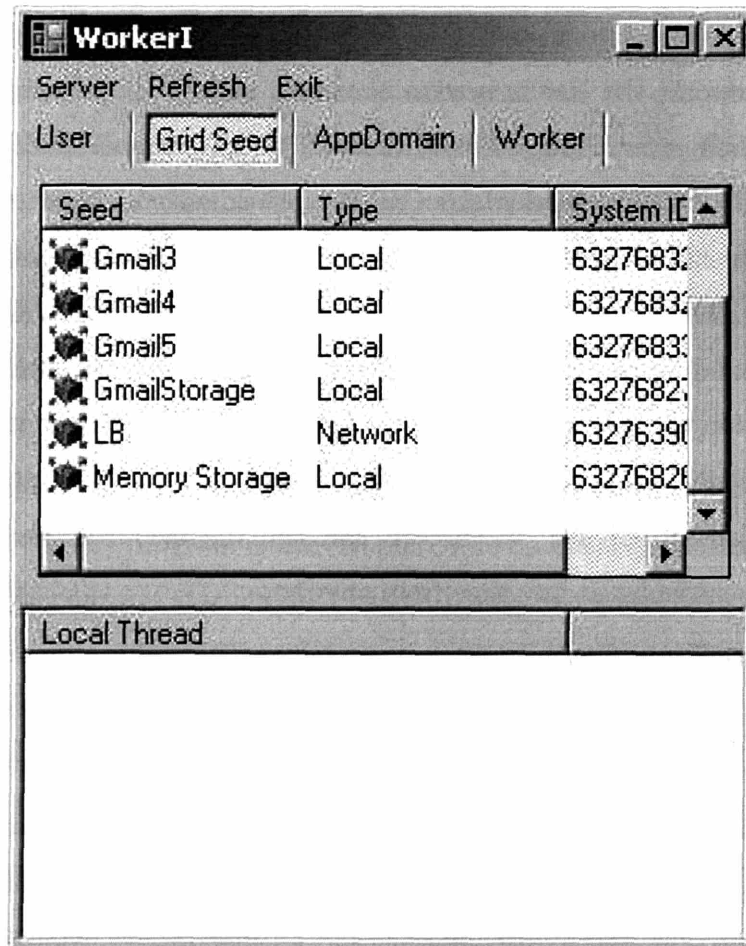


Figure 3-6: The WorkerI interface (Worker manager on Windows).

software package called "SharpDevelop." [8] It has a similar interface and functionality as Microsoft Visual Studio but is free and comes with the full source code. For all these development GUIs, the .Net Software Development Kit (SDK) must be pre-installed [9]

The Realm programming interfaces encapsulated in the GridLib.dll are fully documented. The code examples are also available for reference and a quick start.

3.3.3 Seed Debugging

We acknowledge that debugging a distributed application is different in many ways from the ordinary code debugging task. There are debuggers that come with Microsoft .Net SDK as well as the open source Mono SDK. However, these debuggers are not aware of the existence of the Realm Framework and are not designed for distributed computing. To support the application developers debugging their code under the Realm Framework, we provide a debugger that is specifically designed for the Realm system. It comes with a GUI that looks familiar to Microsoft Visual Studio users. Other than common tasks, such as setting break points, showing values of the variables and stepping forward, the Realm debugger also identifies Worker processes and is able to debug and control individual Worker processes. The debugger is described in detail in Chapter 8.

As we have seen in this section, the implementation of the Realm system achieves the goal of ease-of-use and ease-of-programming. By doing this, the Realm system is likely to attract more attention from people than previous systems.

3.4 Programmability

The Realm Framework provides a rich set of interfaces in a layered model. There are multiple interface layers from the bottom level that deals with the Realm system

[8] SharpDevelop is available at http://sourceforege.net/projects/sharpdevelop/.

[9] The MSDN Web site has all the information for the software mentioned except SharpDevelop in this paragraph.

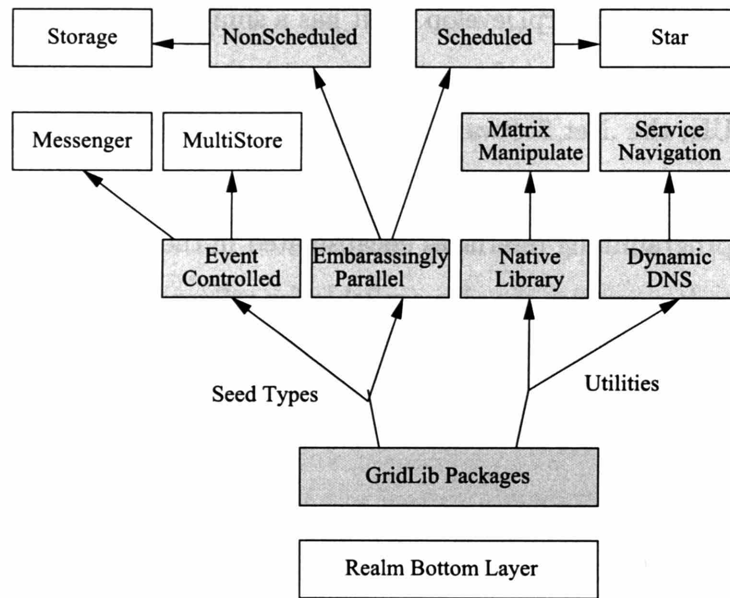


Figure 3-7: The WorkerI interface (Worker manager on Windows).

internals to the top one with interfaces for very specific real-world problems (Figure 3-7). These interfaces allow the application developer to use the Realm system for many problems often seen in engineering or develop extensions to support more features. The previous systems commonly did not consider programmability as a significant issue. For MPI, runtime performance is the top priority. Other than basic message-passing interfaces, there are no problem-specific functions. Grid Computing solutions usually do not consider application-level programmability because they are developed to support general-purpose resource management. The Realm Framework considers programmability as an important factor and tries to support as many use-cases as possible. In this section we generally discuss the bottom and top level interfaces. Detailed descriptions for the intermediate levels are available in the programming reference [10].

3.4.1 The Bottom Level Interface

At the bottom level, the programmers interact directly with the other components of the Realm system. To program in this layer, a programmer needs to consider at

[10] Available in the code package.

least two parties–the Realm Server and the other Realm Workers. The request and command passed to the Seed are all wrapped in a *Messages* object, which is actually a hash table. Multiple requests or commands can be packaged into one Messages object. All requests or commands are in the form of an *EventData* object. From the EventData object, we can extract message ID, source, message type and some other useful information. The bottom level interface does not specify the meaning and the action to take following this information, and so the application programmer has to define the set of actions to take upon receiving a message and supply the implementations of these operations. To talk to the Realm Server and the other Realm Workers, the application developer has three options as described before: the quick message mode, the big object mode and the direct connection mode. For quick messaging, the sender puts an EventData object defined as a quick message into the Messages object and passes it to the Event services (see Figure 3-1) on page 45 on the Realm Server. The Realm Server will tag the EventData object with a serial number and then send it to the receiver. For bigger objects, the sender puts the location of an *object pool* instead of the objects themselves into the EventData object. The object pool, usually another Remoting service initiated by the sender, must be implemented by the application developer and has to consider issues such as garbage collection, object identification and so on. To retrieve the big objects, the receiver must resolve the object pool location conveyed by the EventData object and be able to fetch the object by itself. For direct connection between the sender and receiver, the sender gets the location of the receiver from the Event Sub-system and send the Messages object directly to the receiver.

When a job starts, the Realm Server sends initialization messages to the Seed. These messages must be handled and the application programmer has to carefully decide what to do at start time so that all necessary services and objects have been initialized. Also, at any point of time, the Realm Server may send commands such as *pause*, *resume*, *stop* or *new-node-added*. They should also be correctly handled to maintain a stable Realm runtime environment.

From the above walk-through, we realize that the bottom layer programming

requires high proficiency and extreme caution. At this level the Realm Framework offers little abstraction. For example, the Event Sub-system is hardly a manager of the messages; it is just a information relaying server. Nevertheless, at this layer the application programmer gets great flexibility in customizing the system.

3.4.2 The Parallel Computing Interfaces

On the top layer, two parallel computing interfaces are provided.

The Embarrassingly Parallel Interface

A programming model that is especially appropriate for embarrassingly parallel applications is the master-slave model (see Figure 2-3 on page 30). In this model, the computational job is divided into a sequence of sub-jobs that are independent from each other. The master Worker process dispatches these sub-jobs in-order to the slave Worker processes. Each Worker process does the sub-job and then puts the result into a collection of data or simply returns the result to the master Worker process. The master process receives a *ready* message after the slave process finishes the job and is ready for a next task.

On the master side, the application programmer needs to at least implement GetNextJob() and PutResult() methods so that the Realm system knows what to do next and how to handle the result. Both of these functions pass a system generated sub-job ID as a reference. The application developer uses this ID to identify the result returned from the slave process. By default, the Realm system applies *eager scheduling* [10] discussed later to dispatch sub-jobs to the slaves. The developer may also want to implement the GetScheduler() function if she does not like the default scheduler. Developing a new scheduler takes time and in most cases, the eager scheduler is sufficient in terms of both reliability and performance.

On the slave side, the application programmer only need implement the DoJob() method. The job object from the master side is passed to the DoJob() method. The returned object of this function is the computation result. So by default, the slaves

return the results directly to the master process.

By implementing only three methods–GetNextJob(), PutResult() and DoJob(), an inexperienced application developer is able to write a robust master-slave style distributed program in a very short time. With this high-level abstract interface offered by the Realm Framework, parallel programming is made easy. The application developer does not have to take care of the scheduling nor node management but can just focus on the implementation of the core computational task.

The Communication Based Interface

The communication-intensive parallel computing interface resembles the MPI's functionality with several enhancements (see Figure 2-4 on page 32). The general idea of the communication-based interface is to let the Worker processes communicate conveniently with each other. The communication based interface uses a series of consecutive integers starting from 0 to identify the Worker processes assigned to the job, just as MPI does. The data transfer in MPI is through plain TCP connections. On the sender's side, the data have to be packed in a specific order in an array. The receiver has to unpack the array with caution. The MPI's application programmer has to take significant time to make sure this works in the expected way. An unnoticed error can easily cause the execution to fail. Simpler than MPI's implementation, the Realm Framework's interface uses a facility called *InBox* to manage the received objects. For example, sending an object to Worker process "2" from "1" is as simple as:

```
...
// Identify myself.
if ( this.id == 1 )

// Send object "obj" to "2".
this.SendObj( 2, obj );
...
```

The data sender is never blocked in the Realm System. Receiving an object is very different in our system compared with MPI's implementation. Object Oriented Programming (OOP) shows its potential here. The InBox is provided as a container for received objects. The application programmer defines an InBox for each Worker process that the current Worker process wants to receive data from. Upon receiving the objects, the system delivers the objects into the correct InBox corresponding to the object sender. The InBox is a very powerful tool because its behavior can be greatly modified by changing the InBox's properties. For example, the variable InBox.order specifies how those arrived objects line up:

```
{
        ...
        //First in first out.
        inBox1.order=InBox.FIFO;

        //Only preserve the last object.
        inBox2.order=InBox.LAST;
        ...
        //Oldest object.
        obj1=inBox1.pop();

        //Last object.
        Obj2=inBox2.pop();
        ...
}
```

Also, MPI provides different methods for handling the data transferring, such as blocking and non-blocking connections. The diversity of these functions brings flexibility to application development. The Realm system provides a similar functionality. For example, to block the process when the local computing node has not received new data from one node but leave the other nodes unblocked, we can simply use the following code:

```
{
        ...
        //Blocking receiving.
        inBox1.mode =InBox.BLOCKING;

        //Non-blocking
        inBox2.mode=InBox.NONBLOCKING;
        ...
        //Blocked here if no object.
        obj1=inBox1.pop();

        //Always return immediately,
        //null if no object.
        Obj2=inBox2.pop();
        ...
}
```

Sometimes we have to process the received data immediately but do not want to block the execution in the blocking mode or loop repeatedly in the non-blocking mode. This is hard to achieve in MPI because MPI does not provide event-handling functionality. A possible solution in MPI is to fork off another process or start a thread to listen to the network in blocking mode. Upon received the data we want, the listening thread interrupts the main thread so that the data can be immediately processed. This is rather complicated, and programmers need to exercise caution in writing a correct implementation. Fortunately, in the Realm Framework, application programmers are freed from all these complexities. They can supply as many event handlers as they want to process the data upon receiving them as illustrated in the code fragment below:

```
{
        ...
        //Add event handler.
        inBox1.Ereceiving +=
```

```
        System.EventHandler(receiving);
        ...
}
private void receiving(...,...)
{
        //handle the event.
        ...
}
```

There are some other functions supported by the Realm Framework, such as Globals, Dynamic DNS system and accessibility to the Universal Object Storage. These functions make the communication-intensive computing interface superior to its parallel computing counterparts in terms of programmability. The Realm Global realizes a shared memory over the network. Since the Realm Globals are accessible at any time from anywhere provided the client submits necessary credentials, the end-users can watch the progress of the parallel computing job and retrieve the intermediate results from the Realm Globals. Other than the Global, the Dynamic DNS system enables a universal location for each parallel computing job; the Universal Object Storage provides an Object-Oriented counterpart of the data file. In Chapter 5, we will demonstrate how we can put these capabilties together to solve real-world problems.

3.4.3 The Universal Object Storage Interfaces

The Universal Object Storage supports distributed applications. It offers a distributed solution to keep data objects. A Universal Object Storage Seed acts like a middleman. All such Seeds have a uniform interface for data input and output. They each require an underlying storage medium, often a hard-drive or a database server, so that the data object can be saved in a physical form (see Figure 2-5 on page 33).

As a common requirement, the Universal Object Storage Seed has to implement GetType(), Read(), Write(), List() and Delete() functions. The function GetType() should return the capability of the server, such as writable, deletable, permanent and

any combinations of the supported functions. A reference ID is passed to the Read() function and the data object is returned to the caller; the Write() function takes the data object and a reference ID as the inputs and saves the data object on the backend medium; the function List() returns the reference IDs that match the pattern of the input; the Delete() function removes the data object previously saved on the medium according to the reference ID.

There are some other functions that can also be implemented, including Rename()[11], Hide() [12]and so on. As a minimum, the Read() function must be implemented. In Chapter 6, we will show how to write Universal Object Storage code based on an unusual backend media-an email account and use it as a virtual file system.

3.4.4 The Messenger Interface

The messenger interface takes maximum advantage of the Event Sub-system to build a secure, multi-channeled communication tool kit (see Figure 2-6 on page 34 for the general idea.). By default, 256 channels are created at start-up for each *Meeting* job. Each Messenger Worker process can acquire one or more channels, and later it can release the channels acquired. Each channel can only be acquired by one process, called the channel's head, until it is released. The head of the channel specifies allowed participants and a password to protect the channel if necessary. During the execution of the Meeting job, each process talks to the channel instead of specific users. Among the channels, channel 0 is always acquired by the initiator of the Meeting job, and is always open to everybody, though usually protected by a password. This model operates like a meeting: channel 0 is like the meeting room, the head of channel zero is like the holder of the meeting, and during the meeting, the participants can talk to each other in small groups (channels other than 0). The channel supports any serializable objects, not limited to simple strings. Unlike the communication parallel

[11]Rename() is called by the system when the client wants to rename the object.

[12]Hide() is called by the system when the client wants to make the object invisible. The detailed information of these functions is available in the programmer's reference of the Ream system.

computing interface, the messenger interface uses a user-specified string to identify the meeting participants instead of a system-generated integer.

An application developer needs to implement Invited() and Received() functions when using the Messenger Interface. The Invited() function is called by the system when another process initialized a channel that allows the current process to join; the Received() function is called so that the current process can process the message received from the channel. There are a minimum set of functions that an application developer must know before writing software using the Messenger Interface: OpenChannel() for acquiring a channel, SendToChannel() for broadcasting in the channel, GetChannelList() to get a list of channels and Channel.Release() to release a channel.

We have used this Messenger interface to build a quite complicated application for the supply chain industry. The detailed information can be found in Chapter 7.

3.5 Reliability

Reliability is an important issue especially for a distributed system connected to the Internet. It is also a challenging topic because reliability has many aspects and most of them are hard to tackle. Two major topics in reliability are security and fault-handling.

3.5.1 Security

For a distributed system, runtime security and Internet security must be guaranteed before the system is put into use. The MPI system relies on the security setup of the operating system, without considering it as a significant issue. On the other hand, there are many papers published about Grid Computing security [5], which offer a rich resouce we can refer to.

Runtime Security: Sandbox

We restrict the local resources that a Worker process can use by supplying a Sandbox as a wrapper for the executables. This guards the Worker computer from malicious

code. The Sandbox is actually a .Net application domain (AppDomain). In .Net, we can define security policies for each application domain. The current implementation of the Realm Framework does not supply a hierarchy of policies from which the application programmer can select. The default policy only allows file access to the working directory of WorkerI.exe but allows unrestricted Internet access. Reading environmental variables is also allowed, which provides a quick and easy way for the Worker process to interact with the local computer. Currently the default policy is the only one supported, but expansions are possible.

Internet Security: Secure Remoting Channels

There were two major problems identified when the framework was initially designed. One is that the Realm web services should feature a user authentication and encryption system to protect the Realm community from open attack. A compromised Realm can be a disaster because there are a large number of computers inside a Realm. More seriously, the Worker machine may not be dedicated to the Realm but just sharing its idle resources. There may be very important sensitive information on the Worker computer. Internet security is a top issue in developing the Realm Framework.

Considering the ease-of-use goal of the software development, the network security module should be transparent to the application programmer, in the sense that they should not have to change the way they write code. They should program as if there was no security module active. The .Net Remoting Framework provides a mechanism to allow programmer to modify the default behavior of the messaging channel (Figure 3-8). At the client side, the method call proceeds through a chain of sinks (people can implement their own custom sinks, for example, to perform data encryption) and onto a transport sink that is responsible for sending the data across the network. At the server side, the call passes through a similar but inverted pipeline, after which the call is dispatched to the object. This makes it possible to solve the two problems. The Realm approach is to create a custom sink that can do user authentication and encryption. Often we may only require either one or the

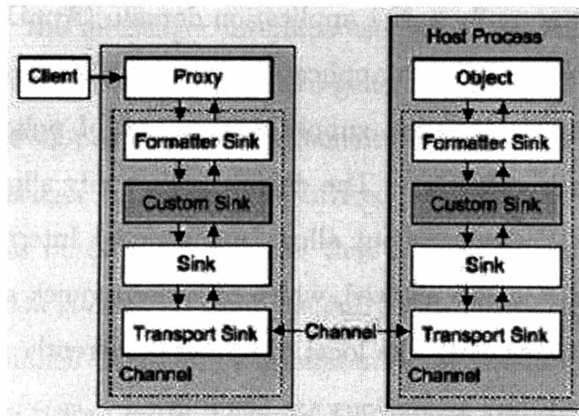


Figure 3-8: The .Net Remoting message chains (from Microsoft).

other due to consideration of communication performance in cases where the network security is not a serious issue. For this reason, the Realm system actually provides two custom sinks, one for user authentication and one for data encryption. When they are used together, the encryption sink needs to sit between the network and the authentication sink so that the authentication information is also encrypted.

Starting with the .Net version 2, .Net Remoting supports a fairly complicated authentication mechanism. By default, the Remoting client needs to be a Windows machine and the authentication goes through the Microsoft Network's Windows authentication service. However, this does not work for Internet users who do not rely on an integrated Windows authentication service. User authentication in the Realm Framework utilizes a very simple authentication sink. Figure 3-9 shows a sample SOAP message that imitates a Web method call when we use the authentication sink alone. The client-side sink inserts a user name and encrypted password as user credentials into the SOAP header section. There is no multiphase hand-shaking process involved in user authentication; the user name and password are provided in one request. The credentials are currently provided in the Remoting configuration files.

The encryption of the data is a little bit more complicated. There is no encryption support for .Net Remoting in .Net version 1.0 and 1.1. At that time the only choice for us was to write an encryption sink used in the .Net Remoting sink chain. The most recent version of the .Net Framework is 2.0. It is the first version to support

```
<SOAP-ENV: ...>
<SOAP-ENV:Header>
<h4:DemoCredentials href="#ref-3"
xmlns:h4="http://schemas.microsoft.com/clr/s
oap/messageProperties" SOAP-
ENC:root="1"/>
<a1:soapaddon id="ref-3" xmlns:a1...>
<username id="ref-6">jrw</username>
<passwd id="ref-7">1-250</passwd>
<encoder id="ref-8">foo</encoder>
</a1:soapaddon>
</SOAP-ENV:Header>
<SOAP-ENV:Body>
..........
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 3-9: The .Net Remoting message with authenticaion in SOAP format.

a secured .Net remoting channel. It is easy to change the default, non-secured TCP or HTTP channel to a secured one by setting the *secure* property to *true* in the remoting configuration file. In this setup, however, we have to supply a valid X.509 [19] file for each computer in the Realm, including the Realm Server and all the Realm Workers. Also, this file is shared by all the Worker processes on the Worker computer. If a malicious process is running on the same machine with other Worker processes, the malicious process can steal the certificate and break the encryption of the communications related to any of the Worker processes on this computer. Moreover, since encryption can often have a strong negative impact on the system's performance, we intend the encryption schema of the Realm Framework to be flexible, so that we can choose the optimal solution suitable for the actual environment. For these reasons, we implemented an encryption Remoting sink that is believed to be the best solution for the Realm Framework.

The encryption sink is inserted in a position close to the network. On the client side, it should be at the end of the send-message chain, while on the server side, it is at the very beginning of the receive-message chain. This is to make sure that the server encryption sink and the client encryption sink are linked together directly

and no communication bypasses the encryption. The protocol resembles how a Web browser establishes a secure connection to a Web server using Secure Socket Layer (SSL). First of all, all data connections are encrypted using symmetric encryption algorithms, such as 3DEC and RC2. These algorithms are ready to use in the .Net Framework. The symmetric encryption means that we use a single key to encrypt and decrypt the data. The problem of using symmetric encryption algorithms for an Internet connection is that both parties of the connection must know this key before they initiate their conversation. The encryption key should not be passed without encryption over the network. Hard-coding the key into the Realm system is obviously an awkward design and in fact offers no security to the system because the Worker program is available to anyone who is interested in the Realm Framework.

The encryption sink uses a mechanism that involves asymmetric encryption to pass the key. For the asymmetric encryption method, a private key is used to decrypt the data and a public key is used to encrypt the data. In this mechanism, both of the Remoting server and the client maintain a hashtable that stores the symmetric encryption key and along with the identity of the communication partner. The client sink also maintains a list of public/private key pairs . When the client is sending a message to the server, the client side sink checks whether a symmetric key is available for the destination server. If there is no key available, the client sink initiates a special request to the server side Remoting service. The message contains a dynamically generated public key for an asymmetric algorithm. This request will be caught by the server-side encryption sink. The server sink then dynamically generates a symmetric encryption key and encrypts this symmetric key using the public key provided by the client. The encrypted key is then returned to the client sink, as is illustrated in Figure 3-10. The client sink decrypts the key and removes the asymmetric key pairs. The communication can then be established using the key and the corresponding symmetric algorithm. All subsequent communications are symmetrically encrypted, illustrated in Figure 3-11.

This mechanism for data encryption is not perfect. It currently lacks support for an asynchronous Remoting request. Fortunately the Realm system is based on

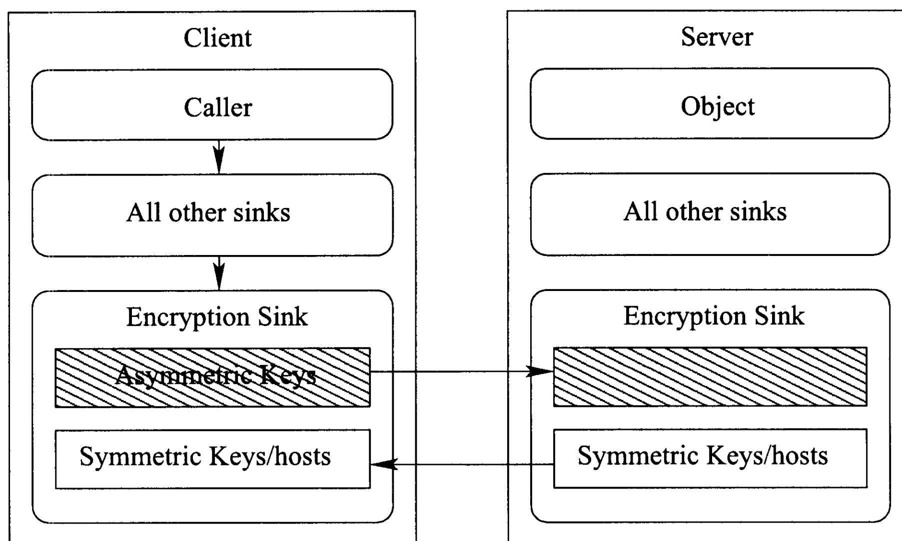


Figure 3-10: The .Net Remoting negotiating step.

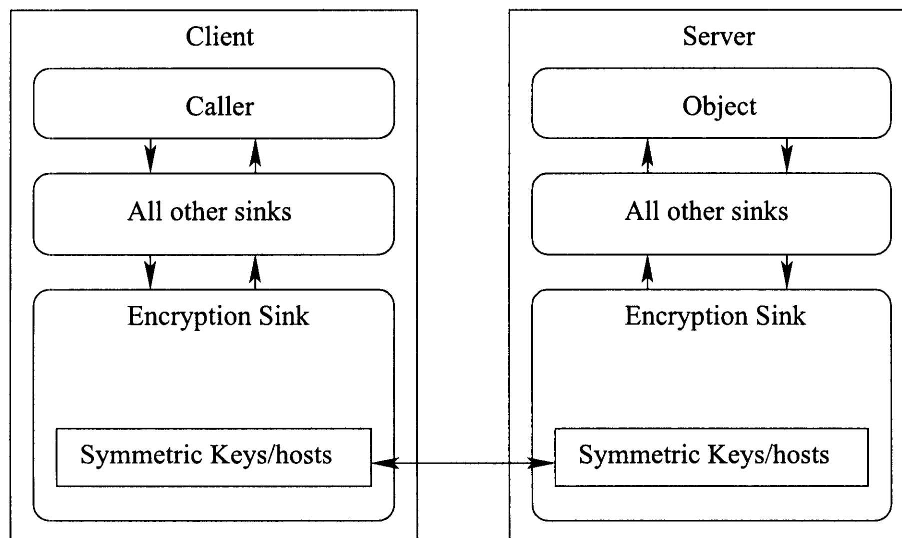


Figure 3-11: The secured .Net Remoting.

synchronous usage of the Remoting services only. The mechanism does not consider potential impostor or Denial of Service (DoS) attacks. For example, a third party can pretend to be the Remoting server to communicate with the client, while at the same time pretending to be the client to communicate with the real Remoting server. Future work may introduce a third party to avoid this impostor attack. The current system may be vulnerable to DoS attack, too. Examinations and further development are needed to evaluate the vulnerability under the DoS attack and protect the system from being compromised.

The Realm system combines the authentication sink and the encryption sink to offer a secure and very flexible communication solution. In the following chapters we show how we can use them in real-world applications.

3.5.2 Fault-Tolerance and Adaptive Parallelism for Embarrassingly Parallel Applications

Although fault-tolerance is hard to implement for all use cases, it is nevertheless necessary for the embarrassingly parallel case, because we want to run the embarrassingly parallel applications over a loosely coupled network, typically a group of computers sparsely located all over the Internet, in the same way SETI@Home and distribute.net do. We also need adaptive parallelism which allows dynamic Worker allocation so that the running jobs can take advantage of the maximum power that the Realm network can provide. We achieve these goals by borrowing an idea from the Bayanihan project.

The Bayanihan project used an extended *eager scheduling* [10] mechanism to support adaptive parallelism and fault-tolerance [31]. In the Realm Famework, it is implemented as the default scheduler accompanying the embarrassingly parallel computing interface mentioned earlier in this chapter. The structure of the scheduler is shown in Figure 3-12. Each sub-job has a "done" flag which is set when a Worker returns the result for that sub-job. The sub-jobs are linked in a circular list. A LastJob pointer keeps track of the the last sub-job that has been dispatched. When

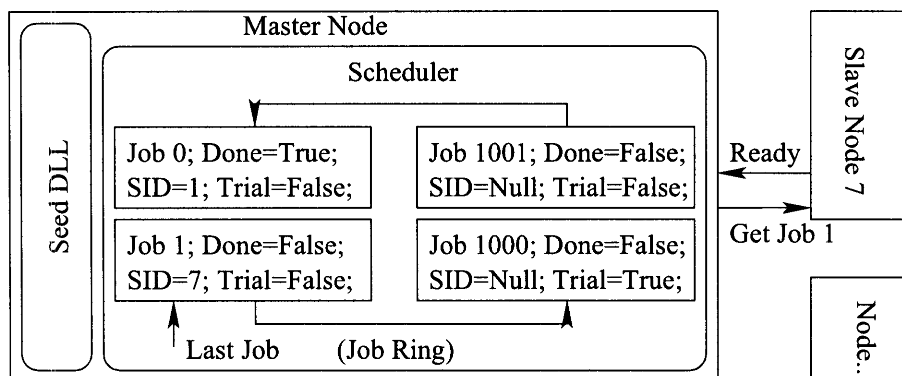


Figure 3-12: The default scheduler used in the Realm system.

a new Ready notification is received from a slave Worker, the master Worker moves the LastJob pointer forward to seek a new sub-job for the slave Worker. If the master Worker finds a new sub-job, it will pass the sub-job object to the slave. Since the job list is circular, if all sub-jobs have been assigned, the LastJob pointer will rewind to the beginning of the list. In this case, the eager scheduling allows the scheduler to seek further ahead in order to find any job that has not been finished yet. This behavior guarantees that slow Workers do not cause bottlenecks because the fast Workers will bypass slow ones and repeat the job. It also avoids hanging the execution because of stalled or dead Worker processes. Further, this eager scheduling does not require a pre-defined set of Workers. Any "Ready" Workers can be assign a new sub-job if available, and any sub-job not finished by a failed Worker will be taken care by other Workers.

The fault-tolerance capability offered by the scheduler is not activated by default because it requires the programmer to implement more functions. The fault-tolerance implementation in the Realm Framework is based on *spot-checking* and *backtracking.* The scheduler keeps track of the handler (slave Worker) of each sub-job. It also maintains a list of *trial-jobs* and the expected result for each of them. In the example shown in Figure 3-12, job 1000 is the trail one. The trail jobs are randomly assigned to the Workers with a trial ratio. For example, a trial ratio can be set to 1% if we want roughly one trial among one hundred jobs. If for the trial job the slave returns a wrong result, the slave Worker is then disabled by the master. All the jobs that have

been done previously by this slave Worker are flagged as not done by backtracking. This mechanism has been proven to work very efficiently [31].

In practice, deterministic results are not available for all problems. For some probability related problems, such as Monte-Carlo simulation, this mechanism is not applicable because we can not offer a list of sub-jobs with pre-determined results. Anyway, computational fault is rarely seen nowadays. We do not activate the fault-tolerance module in most of our applications, in order to shorten the programming cycle and improve the runtime performance.

3.6 Efficiency

3.6.1 Runtime Efficiency

There may be some concern about the efficiency of the CLR and .Net Framework as to whether they are suitable for high performance computing. In particular, the CLR code is a partially-compiled image similar to Java byte code. As for Java executables, the CLR code needs a Just In Time (JIT) compiler to convert the image to the machine-specific code. This would appear to be fairly inefficient. We acknowledge this concern and have done some studies at an early stage of development.

During the early design, the runtime efficiency issue was addressed by using a test case inspired by Matlab [13] . A scheme that manipulates a large volume of data in native code (written in C) was developed and matrix inversion of a dense matrix was tested. The "product" function was mapped into the "Matrix" data type. This data type is seen on the third level of GridLib (Figure 3-7). The usage of this function is:

```
{
        ...
        Matrix m1, m2;
        ...
        m2=Matrix.inverse(m1);
```

[13] Documents for Matlab scripting language and modules are available on web site http://www.mathworks.com/

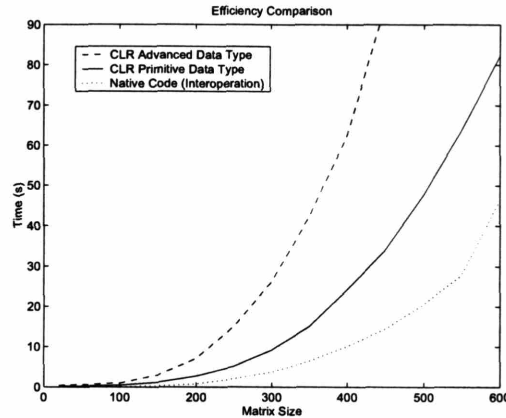


Figure 3-13: Comparison of efficiency.

An experiment to measure the efficiency of the different code was executed. Three types of code were studied: natively compiled code, pure CLR code using primitive types (arrays of double-precision values) and CLR code with advanced data types (linked list of objects). The time cost for different sizes of matrix and different programming methods are plotted in Figure 3-13 [14].

Not surprisingly, the native-code based matrix manipulation is the most efficient one. However, the difference between CLR code using primitive data types and native code is not significant. The time is approximately doubled in CLR code, but still within the same order of magnitude. The CLR code might be regarded as "moderately efficient". The idea of wrapping native functions for large-volume data manipulation does not significantly improve the performance but can be useful for some memory-intensive or computation-intensive jobs. For this reason, we did not actively develop a set of native-code based mathematics libraries to be used in the computational intensive applications. In any case, if extreme performance is truly a concern, the application programmers can still link the Realm runtime system to some high performance packages such as LAPACK [3] without too much work.

[14] Time measured is only the time for product calculation.

3.6.2 Communication Efficiency

There is another efficiency issue. Since the communication channels are usually based on XML/SOAP [15], which tags all data entities, the actual data volume transferred carries overhead. Also, for large XML documents, parsing takes significant time and memory. Unfortunately, in scientific computing, large data sets are very common. These factors reduce efficiency in communication.

To address this problem, we have done some experiments to find out the extent to which the communication efficiency is influenced by use of XML/SOAP. These experiments are discussed with specific application scenarios in the rest of the thesis.

3.7 Conclusion

In this chapter, we presented the detailed implementation of the Realm runtime system. Some utilities other than the core system, such as the DNS system and the debugger, will be described in Chapter 8. We explained why we choose the .Net Framework and C# programming language as the development platform of the Realm system. With the goals of design in mind, we have described the general architecture of the Realm system, and discussed how we achieved the requirements of accessibility, programmability, reliability and efficiency. We can conclude that the design and implementation indeed meet the goals of the project.

We will demonstrate how the Realm Framework can be used to solve real-world problems in the following four chapters.

[15] We do have another option to use the binary data form. However, this approach breaks the accepted XML/SOAP standards.

Chapter 4

The Embarrassingly Parallel Applications

The programming model for embarrassingly parallel computing has been described in the previous chapter. This model, although very simple, can be used in many applications. In this chapter, we first discuss some issues that need to be considered when using the embarrassingly parallel computing interface provided by the Realm system, then proceed to demonstrate its application on two problems: cracking 64 bit RSA code and frame-rendering to make movies using the Persistence of Vision Raytracer (POV-Ray [1]).

4.1 Typical Network Setup

Like SETI@Home and distributed.net, the embarrassingly parallel application for the Realm system runs in a loosely coupled network such as the Internet. The typical setup is presented in Figure 4-1. Since there are far more uncertainties–such as potential attack and disconnection–in the Internet than in a private local network, full security protection is applied, both authentication and encryption. Remoting encryption sinks need to be plugged into the Remoting sink chain for all communications.

[1] POV-Ray is a popular 3D rendering engine available at http://www.povray.org/. It allows using a script to generate 3D images.

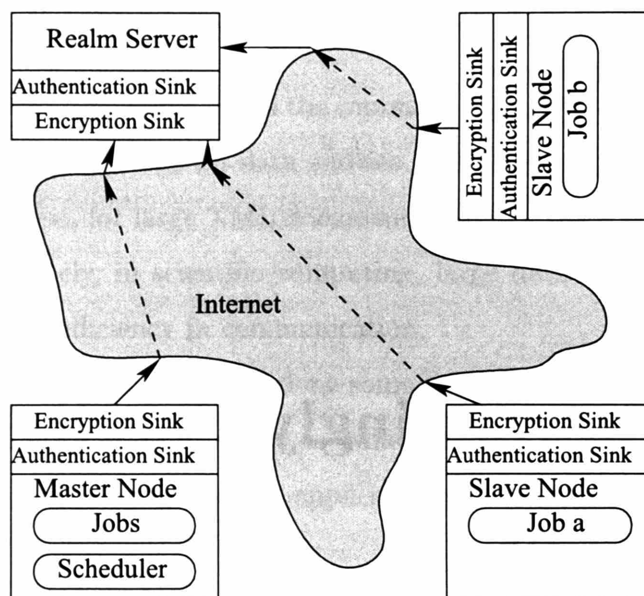


Figure 4-1: The typical network setup for the master-slave style parallel computing.

The first node loading the Worker Seed is the one that initializes the application. It is also the master. The Realm Server loads the Worker Seed into idle Worker computers as slaves. When the slave Worker is initialized, it sends the "Ready" message to the master and begins the computation.

4.2 64 bit RSA5 Encryption Cracking

Cracking an encrypted message is possibly the most popular application that an embarrassingly parallel system would like to try. For example, the distributed.net has a specific project called "RC5 Project". The RC5 Project uses its computing network to search RSA encrypted messages on a daily basis. This popularity is possibly due to the simplicity of the cracking procedure: we simply try all the possible values of the key to decrypt the encrypted message until we find a match.

RSA5 is an asymmetric algorithm. The goal of cracking it is to find out the private key. In our case, the size of the private key is 8 bytes, or 64 bits, so the number of possible keys is 2^{64}, or 1.8×10^{19}. This is obviously too many for a single computer to process. With the embarrassingly parallel computing interface of the

Realm Framework, we can divide the total work into sub-jobs, each of which only processes a small number of key candidates. Each Worker process returns a string of either "FAILURE" or the actually message body if a match is found. The RSA algorithm provides a few bytes at the beginning of the encrypted message. This allows us to tell whether a match is found by comparing the decrypted message with the encryption code itself. The master node prints the number of keys tried and the time consumed. In our test, the original message being encrypted is, "The quick brown fox jumps over the lazy dog."

Now we study the computing speed under the Realm system. First, we compare the ideal speed and actual speed we get. The partition size[2]is 2^{16}, 2^{8}, and 2 keys. For example, in the case of the 2^{16} partition, the master Worker increments a 6-byte unsigned integer by 1, and passed these 6 bytes of data to an individual slave. The slave Worker process appends 2 bytes to the end of the received data to form an 8 byte key. The performance of different setups of the network is also compared in two other charts-the original .NET Remoting TCP/HTTP channel with optional authentication and encryption support. As expected, the partition size has a significant impact on the overall speed. If the partition is small, the slave Worker processes need to spend a larger portion of time in communicating with the master Worker process (Figure 4-2. The master is also kept busier in handling requests with a smaller sized partition than with a larger partition. We also find that for a reasonable partition size of 2^{16}, what kind of Remoting setup we use is really a trivial issue, as shown in Figure 4-3, because the time used for communication only takes a tiny share of the total time. For a small partition size, however, the Remoting style used in the application has to be seriously considered, as is suggested by Figure 4-4.

The last thing to clarify here is that we have actually never truly discovered the original message from the encrypted one because the the total computational work to crack a 64 bit key is too huge.

[2]A partition contains the keys that should be tried by a slave node in one execution. The partition size refers to the number of keys in the partition.

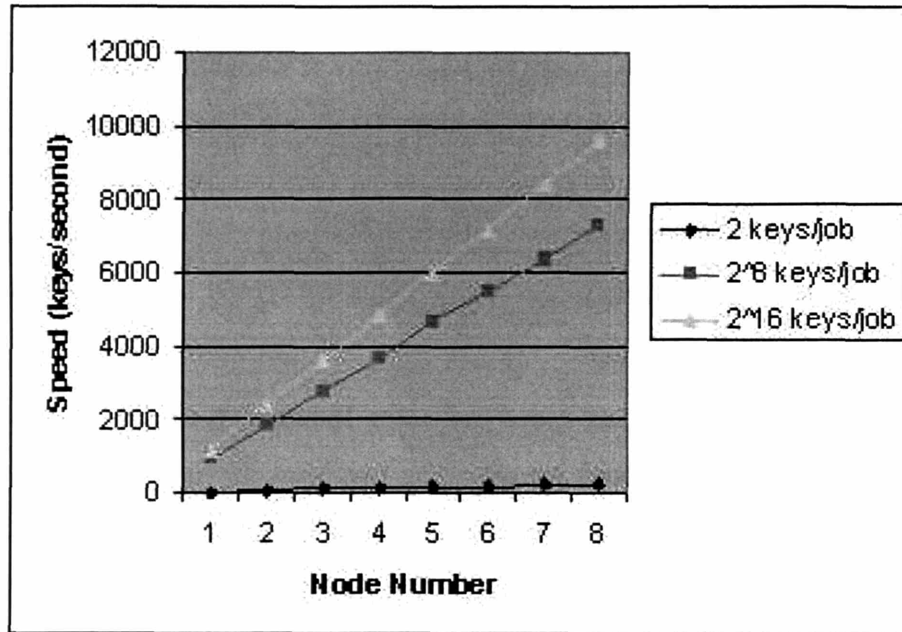


Figure 4-2: Performance for different partition size.

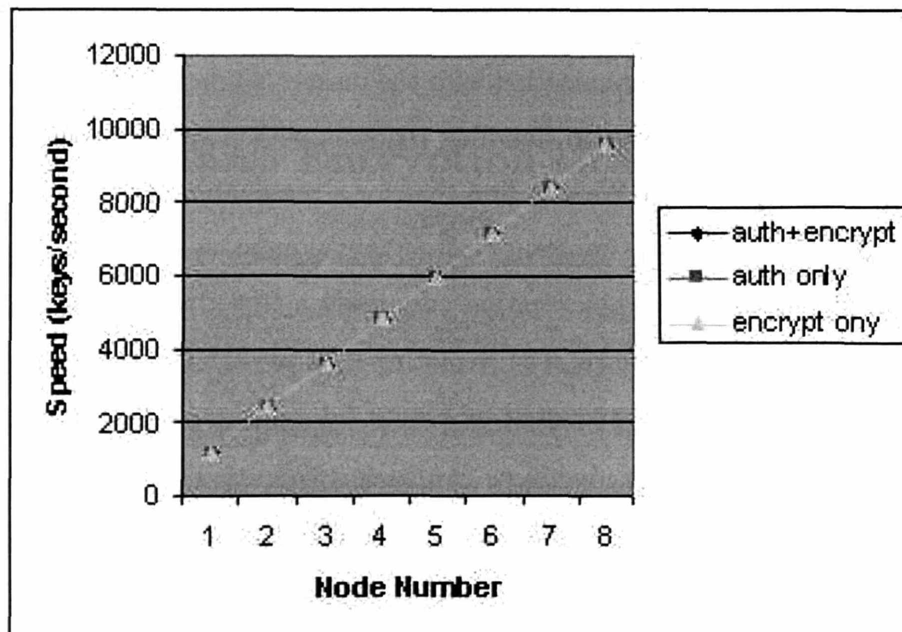


Figure 4-3: Performance for different network setup (partition size=2^{16}).

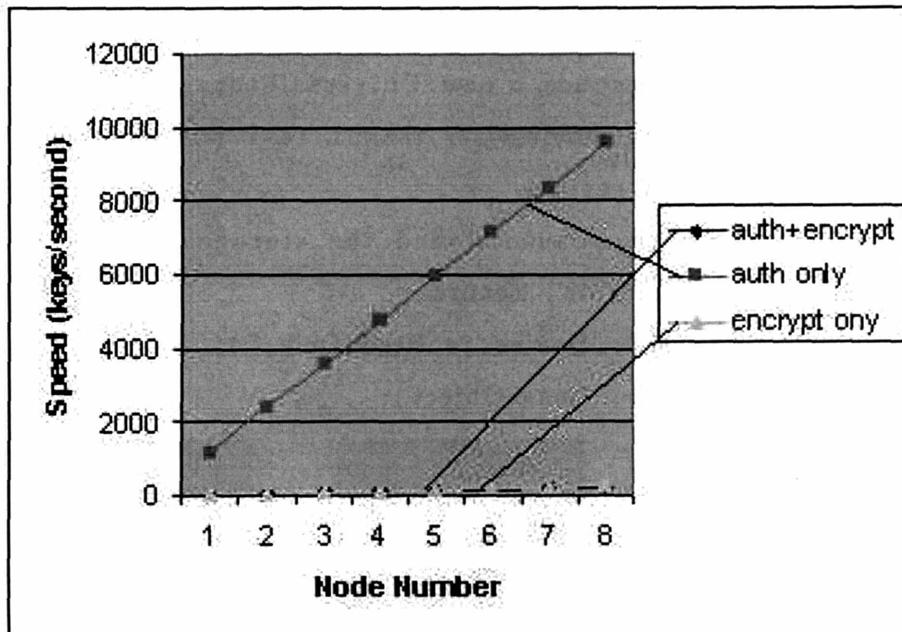


Figure 4-4: Performance for different network setup (partition size=2).

4.3 Movie Making Using POV-Ray

In this section, we demonstrate how we make a movie using the POV-Ray software. The role of POV-Ray is to render a series of image frames. In this test case, we used a group of FreeBSD machines instead of the computers running Microsoft Windows to test the capability of the Realm Framework running on platforms other than Windows. Due to the limitation of the Mono Framework we chose as the replacement for the .NET Framework, we did not chain up any Remoting sinks. The Realm Server was still running on Windows.

Also, to avoid overwhelming the master Worker with a pile of large images, the slave Worker did not return the image to the master Worker. Instead, it saved the image to a Universal Object Storage node and returned a string specifying whether it was successful in rendering an image. To achieve this, the Seed *MovieMaker* uses the Universal Object Storage functions like this:

```
{
...
```

```
        //Initialize the proxy object by its location.
        UniversalStorage uos = new UniversalStorage(
                "Frames.moviemaker.realm1.iesl.mit.edu");
        if (uos==null) return
                "Failed: Cannot found the storage.";
        if (!uos.IsWritable) return
                "Failed: Device is Read-Only,");
        uos.Write(JobID, ImageObject);
...
}
```

In the above sample, the image is saved in the Universal Object Storage at the location "Frames.moviemaker.realm1.iesl.mit.edu" with the JobID as the key. Later on, the user can retrieved these image frames by using the JobID as a reference.

Each sub-job object the master sends to the slave is actually a text message containing a POV-Ray script. The master calculates the parameters for the job and embeds them into the script. On the slave-Worker side, this script was passed to the "povray" program for image processing under the current working directory. Upon finishing the processing, the slave process picked up the image file generated and sends it to the Universal Storage node. The generated frames are shown in Figure 4-5. We did not compare the performance of the same code on Windows because the image processing by POV-Ray is the most time-costly step, which is irrelevant to the Realm's performance.

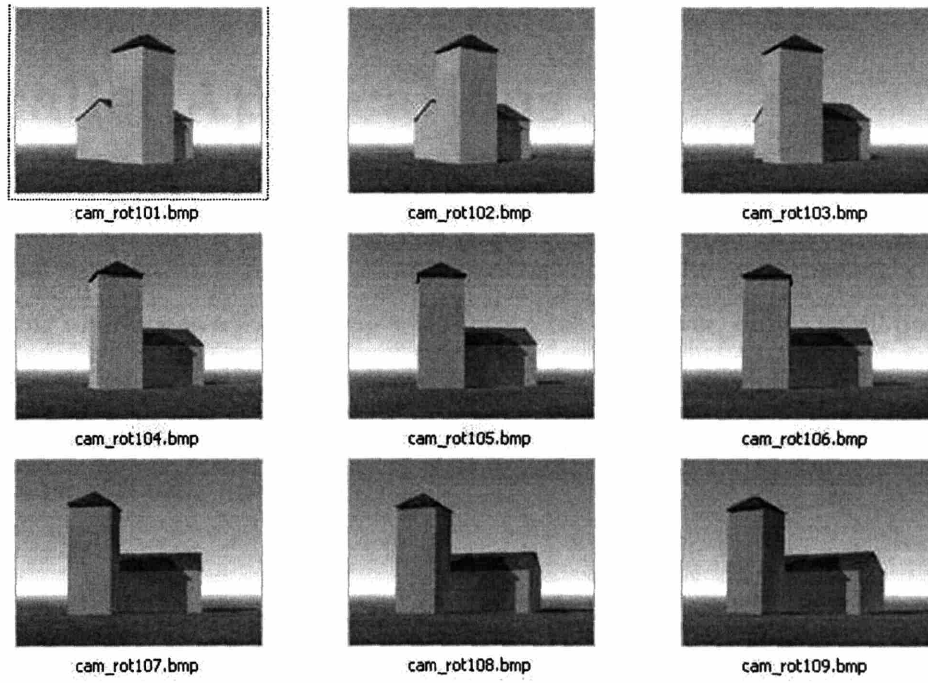


Figure 4-5: Movie frames generated by the distributed POV-Ray based on the Realm Framework.

Chapter 5

The Communication-intensive Parallel Applications

In the last chapter, we presented the applications that use the embarrassingly parallel computing interface to solve the real-world problems. However, only a small portion of the problems can be perfectly partitioned into independent sub-jobs. In this chapter, we demonstrate how we solve more complicated problems with the communication-intensive parallel computing interface.

5.1 Typical Network Setup

Unlike the master-slave network topology we used in the last chapter, the network setup for the more general parallel applications is less flexible. The communication; intensive parallel interface is not adaptive–once a group of Workers have been assigned to the job, no other Workers can be added, and no participants can be removed. For this reason, the connections inside the Realm should not be on the unreliable Internet. Also, the number of the Worker machines is most likely much smaller than in the master-slave style computing. A private LAN is always preferred in this case because we can save processing time by benefiting from high-speed data transfer rates, and by removing the encryption or even the authentication Remoting sinks. The following applications are tested on an 8-node co-located computer cluster with a mixture of

Windows and FreeBSD machines. They are significantly more complicated than those applications we have in Chapter 4, and so we discuss them in more detail.

5.2 Simulate the Mixture of Fine Solid Particles and Fluid Using Lattice-Boltzmann Method

This problem comes from the oil industry. In recent years, the price of fossil oil has been dramatically increasing. This makes mining the "sand oil", which was once not profitable considering the cost to mine it, a very potentially profitable source of oil production. The sand oil is actually a mixture of petroleum oil and sand particles. Simulating its movement under pressure helps people to optimize the locations of wells and the other technical parameters of their operation.

Simulations for fluid dynamics is commonly very computationally intensive. The application that we discuss here is a simulator based on the Realm's communication-intensive parallel computing interface. It simulates the motion of the solid particles and the fluid under various conditions. To simplify the problem, we assume the solid particles are infinitesimal and apply a very simple mechanism to handle their interaction with the fluid.

5.2.1 The Lattice-Boltzmann Method

Scientists and engineers usually describe a fluid flow by introducing a representative control-volume element on which macroscopic mass and momentum are conserved. This leads to a "macroscopic" mathematical model, governed by the Navier-Stokes equation [9]. More recently, "bottom-up" particle methods, such as Lattice-Boltzmann, have been formulated based on a microscopic model derived from statistical particle mechanics [6, 7, 29, 32]. The motion of the fluid particles is described by particle velocity distribution functions valid at each element (lattice-grid point). The method calculates physical variables such as velocity and pressure by tracking the probability distribution of fluid particles moving in different directions.

The Lattice-Boltzmann equation is given by:

$$f_i(\bar{x} + \bar{e}_i \Delta t, t + \Delta t) = f_i(\bar{x}, t) - f_i^{eq}(\bar{x}, t))$$

where the f_i is the concentration of particles that travels with velocity $\bar{e}_i$. With the discrete velocity $\bar{e}_i$ the particle distributions travel to the next lattice node in one time step Δt. The relaxation parameter τ determines the kinematical viscosity ν of the simulated fluid, according to

$$\nu = \frac{2\tau - 1}{6} dx^2/dt$$

The discrete velocity vectors in 2D have the following value and direction:

$$e_i = \begin{cases} dx/dt, i = 0, 1, 3, 5, 7 \\ \sqrt{2} dx/dt, i = 2, 4, 6, 8 \end{cases}$$

where $c = dx/dt$ is the ratio between lattice size and time step.

The equilibrium distribution function f_i^{eq} is calculated as

$$f_i^{eq} = w_i \rho (1 + 3\frac{\bar{e}_i \bar{u}}{c^2} + \frac{9}{2}(\frac{\bar{e}_i \bar{u}}{c^2})^2 - \frac{3}{2}(\frac{\bar{u}\bar{u}}{c^2}))$$

where $w_0 = 4/9$, $w_1 = w_3 = w_5 = w_7 = 1/9$, and $w_2 = w_4 = w_6 = w_8 = 1/36$. The macroscopic density ρ and velocity vector $\bar{u}$ are governed by the distribution functions

$$\begin{cases} \sum_{i=0}^{8} f_i = \rho \\ \sum_{i=0}^{8} f_i \bar{e}_i = \rho \bar{u} \end{cases}$$

The Lattice-Boltzmann equation simulates a slightly compressible fluid; consequently, the fluid pressure p is given by $p = c_S^2 \rho$ where the speed of sound is given by $c_S = \frac{c}{\sqrt{3}}$.

5.2.2 Simulate the Fine Particles

The solid particles in our context are modelled is infinitesimally small, which means we can omit their sizes compared to the simulation area. Also we assume they are of equal weight and there is no energy damping in collision. These three simplifications significantly reduce the complexity of the problem. Without these simplifications, the movement of particles is usually simulated by the Discrete Element Method (DEM) [8]. However, coupling the DEM and Lattice-Boltzmann methods under the parallel computing environment is a great challenge. In the current stage, we separately handle the DEM and Lattice-Boltzmann simulations using the Realm Framework. Coupling them is proposed as a future work.

The "zero particle size" assumption allows us to simply use one lattice point to characterize the spacial properties of a solid particle. This is not only beneficial to the simulation algorithm itself, but also to the partitioning of the domain. We do not have to do any special handling for the solid particles in domain decomposition, since the particles do not have any other spacial properties other than a location represented by the lattice points.

The "same weight" and "zero energy lost" assumption further frees us from the need to calculate interactions among the particles. This is simply because from Newton's law, after two identical spherical objects collide without energy lost, they will each assume the momentium of the other. In practice, we do not have to consider the collision between two particles, because there is nothing changed except a swapping of particle numbers. It saves us from the neighbor sorting and sophisticated mechanics that have to be treated in DEM.

It is true that these assumptions may not be applicable for all cases. Nevertheless, this does help us to explore the potential of the Realm Framework in solving this kind of problem.

The particles are mainly driven by the fluid pressure and the shear stress due to the fluid viscosity. The effects of gravity and buoyancy are omitted in our case.

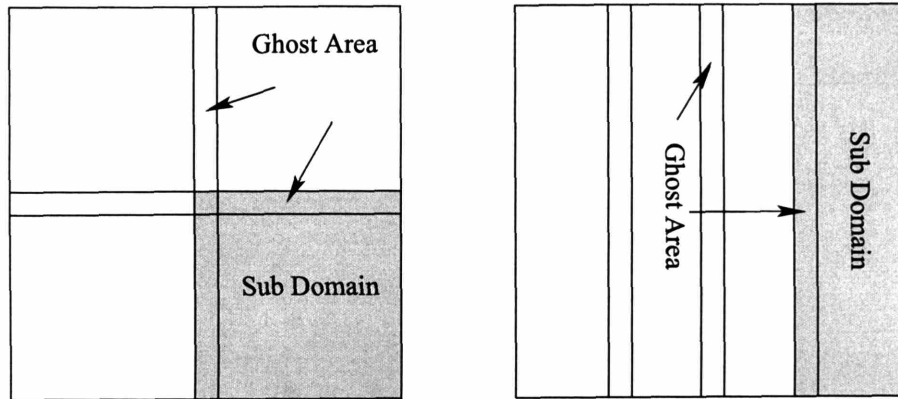


Figure 5-1: Performance for different network setup (partition size=2.

5.2.3 Domain Decomposition

The computational domain is partitioned into sub-domains to be processed by different worker machines. There are two methods of decomposition over the space of the simulation, 1 dimensional or multi-dimensional (Figure 5-1). In the 1D partition, the whole domain is split in one direction, either vertically or horizontally, whereas in the multi-dimensional partitioning, the domain is divided into two directions for 2D simulation or three directions for 3D simulation, resulting in a few rectangular or prismatic sub-domains. Previous work [32] showed that multi-dimensional decomposition is not superior to 1D decomposition for the lid-driven cavity problem[18] in a 2D domain. In this simulator only 1D vertical decomposition was studied, but this is sufficient to illustrate the important points.

The simulation program maintains a *LatticePoint* class. At the end of each time step when the computed value at the "ghost" [1] LatticePoint need to be exchanged, each worker simply sends out the LatticePoints objects within the ghost area to its two neighbors. Any solid particles, instantiated as *Particle* objects, are also transferred to the neighbors if they are inside the ghost area. In the simulation, we use blocking and the first-in-first-out InBox property (described in chapter 3) to receive "ghost" LatticePoints and Particles.

[1] In parallel computing, we often use a "ghost" area to link two adjacent partitions. The ghost area is the overlapping part of the two partition.

Figure 5-2: 2D Lid-driven flow.

5.2.4 Results

The results are stored on the Realm Server in the form of Globals. The Globals are different from the inter-Worker communication data since they are accessible from the Internet via Web services and SOAP messaging. The Realm system gives users the power to access data from virtually everywhere on the Web in a machine-independent manner. In the simulation here, a real-time Graphical User Interface (GUI) program was written to monitor the velocity distribution in an intuitive way. Figure 5-2 shows a simulation for the pure fluid without particles. It simulate the famous lid-driven flow [18] which has been thoroughly studied. In the lid-driven cavity flow simulation, the boundary meets the no-slip condition. The Reynolds number is 1000 and top lid velocity is 0.2 m/s. In Figure 2, velocity is profiled from low to high with colors from blue to purple. The lattice size is 400 by 400, divided into 8 sub-domains. The gray lines denote the sub-domain boundaries. The lid is at the top boundary.

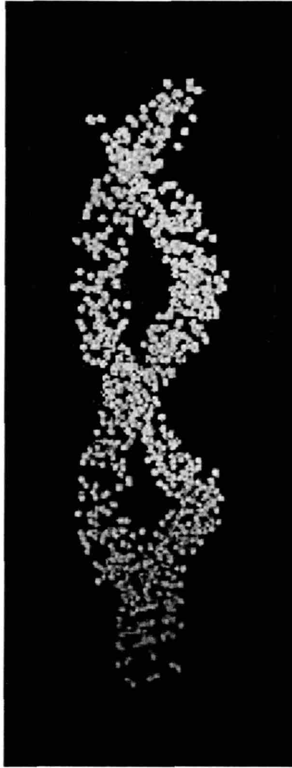


Figure 5-3: Blood flow in a branched blood vessel.

Our final goal is to achieve 3D simulation with a high density of solid particles. For this we have developed a 3D simulator and tested it on three situations: blood flow in a branched blood vessel (Figure 5-3), a thick mixture in a blender (Figure 5-4) and particles accumulating around a hole under fluid flow (Figure 5-5). All of these simulations have been tested on an 8-node cluster with the management of the Realm system. We still follow the 1D partitioning for these cases. The partitioning planes are perpendicular to the longest axis of the containers.

Figure 5-6 shows a comparison for the performance of the code when applied to the blood vessel flow simulation. Theoretically the time cost[2] should converge to zero when more computer nodes are added in. Due to the communication cost, however, this is not true in practice. From the figure we can roughly identify a limit at around 10,000. This should be the actual communication cost for the simulation,

[2]We evaluate the time spent on one execution of the simulation program. It is not the sum of all the CPU times spent on the nodes.

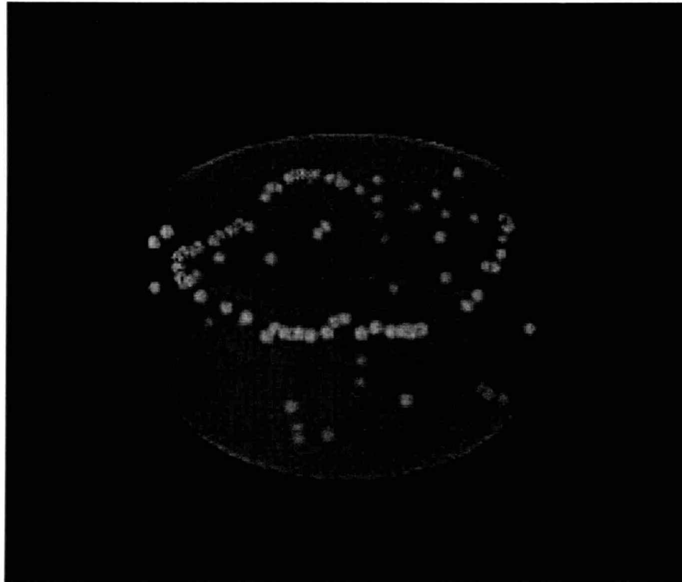


Figure 5-4: Thick mixture in a blender.

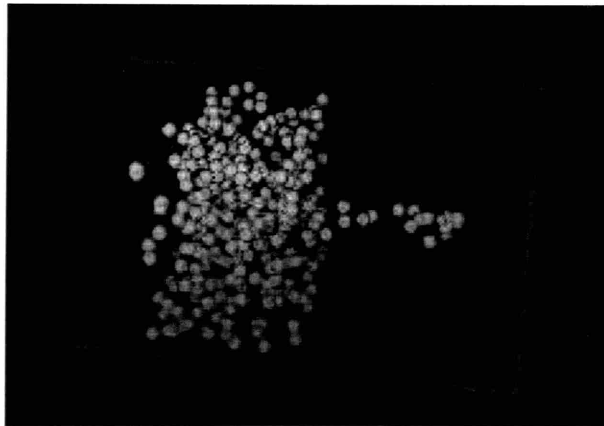


Figure 5-5: Particles accumulating around a hole.

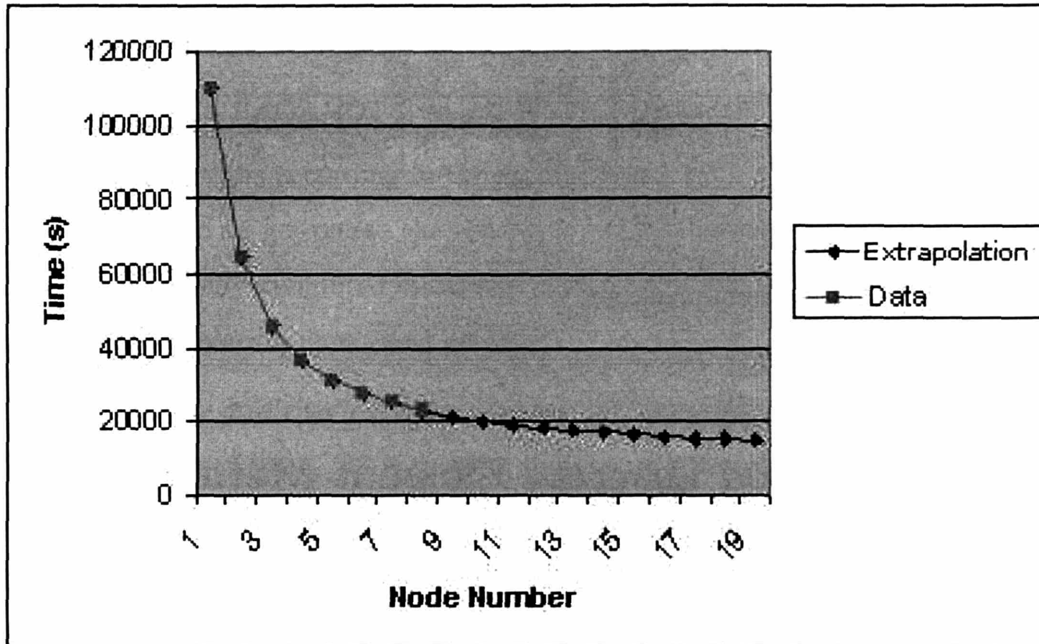


Figure 5-6: Convergence of the time cost.

because we use 1D domain decomposition and the number of ghost lattice-points are identical for each sub-domain regardless of the total number of nodes. Notice that if the total number of nodes is very large, adding more nodes will only burden the Event Sub-system and the total time cost is expected to increase.

Since the performance is an important consideration, we also investigated the overall runtime efficiency under different Remoting configurations [3] so that the application programmer can have an idea of how to get maximum throughput from the Realm system. We evaluate the time-to-finish of the simulation using Remoting HTTP or TCP channels, with binary or SOAP formatter. The results are compared in Table 5.1. We measured a best performance for the TCP binary channel, while the HTTP SOAP channel lags behind all the others. The differences between different formatters are much greater than the differences between protocols, which means the message format is more influential than the protocol. The SOAP message format introduces a significant overhead, consistent with what we hypothesized in Chapter 3. For this reason, we recommend the binary channel for those applications involving

[3]They are done without the Authentication and Encryption sinks.

Table 5.1: Communication Channel performance comparison (Blender, 8 nodes).

	HTTP SOAP	HTTP Binary	TCP SOAP	TCP Binary
Time Cost (s)	97030	47990	94450	32110

significant inter-node communications.

5.3 Distributed Discrete Element Method

Meshless, or particle-based methods, such as the Discrete Element Method (DEM), are computationally intensive for large-scale problems. Distributed computing has the potential to alleviate the limitations of using a single computer. The limitations are commonly in computing power and storage, the resources that can be aggregated by the Realm Framework. We studied the implementation of a distributed solution for DEM problems. The following overview describes some aspects of the solution.

The computational challenges associated with DEM can be broken down into two phases: *contact detection* and *contact resolution.* Contact detection first needs to find closely placed objects and then identify whether two neighboring objects are in physical contact. The contact resolution phase calculates the forces due to the contact and integrates them to update the momentum of the objects. A sorting step in contact detection is necessary. Usually the neighboring sorting uses a much simpler method to find neighbors than detailed contact identification. This avoids the $O(N^2$ cost associated with the all-to-all check using the expensive contact identification method.

Fortunately we do not have to deal directly with these issues. We use the C# based DEM software developed by Dr. Scott Johnson [21]. This software package is designed for a single computer. It solves the general DEM problems with an easy to use interface. My work focuses on the parallelization of the software. For this reason I only discuss the issues that are related to the parallelization.

5.3.1 Domain Partitioning

It has been a common practice in parallel computing that a 3D space is divided into overlapping partitions according to the *special* features of the simulation domain. A ghost area is used to guarantee the coherence of the computing across sub domains that are individually computed on worker nodes. This method was also used in the fluid-particle simulation mentioned above.

After carefully studying the characteristics of the DEM problem, another partitioning method was used. This method divides the physical objects, usually solid particles, into distinct groups. Each group contains a number of objects. One Worker process is assigned to each object group. No spatial shadow area is needed. Instead, particles are tagged as "tangible" if they have neighbors with particles in another group. So this mechanism is highly related to the result of the neighbor sorting. For each time step, the contact identification involves an additional procedure to check the "tangible" tag of each particle. If the particle is tangible, the Worker fetches information from other groups that host those particles that are the neighbor of the tangible particle. As you can see, we do not have to manually place a fixed ghost area, nor do we need a complicated partitioning method. In practice, the neighbors of the tangible particles are fetched from the other Worker processes on a an as-needed basis.

There are several reasons to choose this parallelization scheme. Firstly, the shapes of the space can be extremely irregular, for example, a few convoluted blood vessels. The boundaries may move or even change shapes. Human intervention in the simulation is often unavoidable if space-based static partitioning is used. Secondly, for DEM, static spatial partitioning may result in a bottle-neck situation in which most of the particles are jammed into a very small number of computing nodes. This may cause the execution to fail or waste significant computing power on the idle nodes. Moreover, dividing the space into regular sub-spaces creates problems when the size and shape of the objects are very irregular, because big objects can cover more than one sub-space in some extreme cases. Lastly, the nature of DEM allows the contact-

based partitioning because it relies on discrete particle interaction.

There are some issues raised by this approach. The first one is when and where we should do the neighbor sorting. A naive way is to have a dedicated node to do the neighbor sorting after a number of time steps. In this way we can reuse the non-parallel code for neighbor sorting. However,the neighbor sorting could take a significant portion of time for even a small number of particle objects in the 3D case. For this reason, neighbor sorting should also be parallel. At the same time we should keep the communication cost small. This is challenging because the neighbor sorting should be done globally and the consensus is hard to make without the information of all the particle objects.

Let us first study a partitioning method that can be used in this parallelization scheme. A *perfect partitioning* divides the particles into equal sized groups with minimal dependence. This reduces both load balancing and communication cost. The partitioning problem, for the simplest case of dividing the particles into two groups, can be reduced to a typical problem of Graph theory. If each contact pair is deemed an undirected edge, then the particles form a graph. The partitioning problem is then equivalent to the Minimum Graph Bisection problem [14], which reads:

A bisection of a graph $G = (V, E)$ with an even number of vertices is a pair of disjoint subsets $V1, V2$ of equal size. The cost of a bisection is the number of edges $c = (a, b) \in E$ such that $a \in V1$ and $b \in V2$. The problem of Graph Bisection takes as input a graph with an even number of vertices and returns a bisection of minimum cost.

Unfortunately, the Minimum Graph Bisection problem is NP hard, meaning there is no reasonably efficient algorithm to solve it, especially for the case of DEM in which a large number of vertices (particles) are common. Approximate solutions are available[12]. However, these solutions are hard to parallelize. We therefore propose another method, following the procedure below.

1. Each Worker process holds around the same number of objects.

2. Each Worker knows a pre-determined *reference axis.*

3. At the beginning of a super time-step [4], each Worker computes the location of the projection for each object. These locations are pre-sorted into discrete bins. Say, if the bin size is 3, then an object at 4.3 will be sort into a bin that covers 3 to 6 (Figure 5-7). It is possible that one object may belong to more than one bin depending on its size.

4. Each Worker sends all its bins to all other Workers.

5. After a Worker receives all bins from all other Workers, it then counts the number of objects from the first bin and extracts the IDs of the objects that should be assigned to it in the following time steps. For example, if the number of objects is 10000 and there are 10 Worker nodes, the Worker number "3" will have the objects from the 3001st to 4000th according to the reference axis. For an object covering more than one bin, only the first bin is considered.

6. Each Worker then compares the ID of the objects received from the last step and the objects that are already held locally and fetches the missing objects from the other Workers.

7. The objects neighboring with non-local objects [5] are marked "tangible" and their external neighbors are those neighboring non-local objects.

8. The computation in normal time-steps is then based on the local neighbor sorting with the special handling of the tangible objects.

In short, this parallelization mechanism considered as a reduced CGrid method[37] in 1 dimension, can be thought of as a global neighbor sorting algorithm and load-balanced partitioning with irregular ghost areas. The global neighbor sorting and partitioning are done by individual Workers in parallel with low communication cost. In addition, the global sorting results can be reused in local neighbor sorting. In fact,

[4] a time-step that requires re-partitioning and global neighbor sorting, usually less frequent than the normal time-step that computes the forces and momentum

[5] Non-local objects are those objects outside of the current partition.

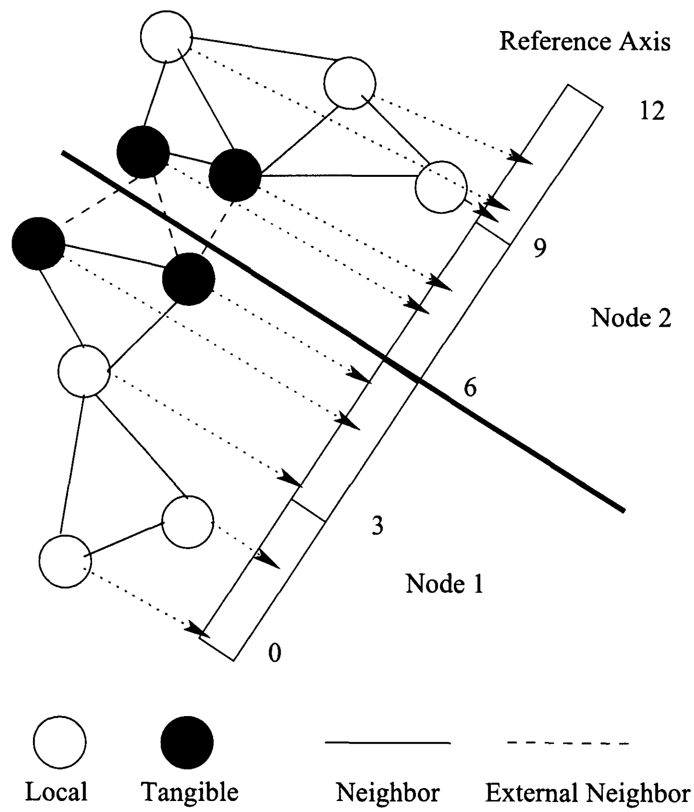


Figure 5-7: The 1D global partitioning using a reference axis.

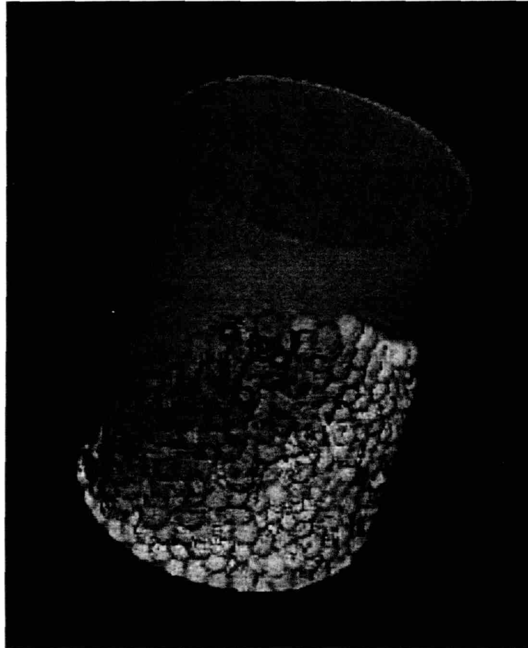


Figure 5-8: Spheres in a cynlinder (512 spheres).

for local sorting we do not use the complicated 3D CGrid algorithm. Instead, we simply look for the overlapping of the bounding boxes for adjacent objects identified by the reference axis. The disadvantage of this method is we must find the reference axis for each super-step. Calculating an optimal axis requires the consideration of two things: how to do that in parallel, and how to guarantee consensus about the reference axis for all Workers. For the current stage, we manually set the reference axis. Fortunately many problems allow us to do this intuitively, for example, if the problem space is inside a cylinder or a torus. Further study should seek a better solution for finding the reference axis automatically.

5.3.2 Results and Comparisons

We have use the parallelized DEM software to simulate solid spheres in a rotating cylinder 5-8. Different partitioning methods are compared in Table 5.2: fixed partitioning along Z [6], fixed reference axis along X, fixed reference axis along Z and

[6] In the implementation, we can set a different policy for allocating the objects into Worker nodes. Instead of using equal-sized partitioning, we can partition the objects by their absolute position on the reference axis. This still guarantees consensus and should be roughly equivalent to the traditional

Table 5.2: DEM Parallel Partitioning performance comparison (2048 spheres, 8 nodes).

	Fixed (Z)	Fixed Ref (X)	Fixed Ref (Z)	Rot Ref
Time Cost (s)	30200	32080	21660	19150

rotating reference axis along the cylinder's central axis. Clearly, partitioning along X axis is a bad choice because in this case the number of external neighbors are large. It is so inefficient that even the fixed partitioning solution is superior. When we have a better reference axis, the performance is greatly increased. From this table we can conclude that our partitioning method indeed improves performance, provided that a good reference axis is provided.

All in all, the Realm system provides a platform so that we can try complicated approaches in solving engineering problems. Without the Object Oriented communication-intensive parallel computing interface, these problems could not be solved in such a short period of time.

method in performance.

Chapter 6

The Storage Applications

In Chapter 4, we presented a distributed movie-renderer that relied on a stand-alone Universal Object Storage node. In this chapter, we briefly describe how to develop a Universal Object Storage system. Two sample applications are demonstrated here: one simply uses the file system as the backend storage mechanism, and another one takes advantage of the storage space offered by Gmail [1]. We then show a simple program that saves and retrieves files using the Universal Object Storage interface.

6.1 Typical Network Setup

We use the storage node as a universal data backup facility. So far it has not been able to support true distributed storage, where multiple nodes can be used to backup a single object [2]. The typical usage at the current stage is to run the Universal Object Storage Seed locally. Whether the authentication and encryption Remoting sinks are needed depends on the network the node is connected to. If it is on the Internet, encryption and authentication are required to protect the Worker from being exploited by malicious connections.

[1] Gmail is a free Web Email system developed by Google. Its storage space is more than 1 gigabytes.

[2] The GridFTP, coming with Globus Toolkit, is able to split a file into pieces and store them over a number of grid points

6.2 Saving Data on The Local File System

The first example uses the local file system as the backend storage media. The idea is fairly simple. Upon receiving the object, the storage seed serializes it into an XML file with a name identical to the object reference. When the Read() function is called by a remote process, the seed program checks the reference received, de-serializes the object, and returns it to the caller.

The serialization should not be a problem because any object transferred through the Remoting channel must be serializable. However, there is a serious issue if we send the object directly to the Universal Object Storage: if the receiver, the storage seed, cannot resolve the data type of the object, the data object will never be accepted. This is very common in Remoting programming. For example, if the programmer defines a new class named "Particle" in a parallel computing application and wants to save Particle objects in a Universal Object Storage node, she cannot just pass them to the storage node because the implementation of the Universal Object Storage seed does not necessarily include the same definition of the Particle class. To solve this problem, the Realm Framework provides a wrapper class *CommonObject*. The CommonObject class contains a function named *CommonObject.Is()* [3] which serializes the object passed to the function into an XML blob [4]. The recovery function is called *CommonObject.Get()* which de-serializes the XML blob into the original object. The object passed to the storage node in this case has the type of CommonObject which can be resolved by the storage node.

6.3 Save Data into Gmail Account

This example shows how versatile the Realm Framework is. We connect the Realm Framework to a Gmail account by the "GmailStorage" seed program, and use the big email storage space as the underlying storage media.

To achieve this, we first identify what are the possible ways to read and write data

[3]This function should be called by the client of the Universal Object Storage.

[4]The XML blob can be further compressed to save memory and communication cost.

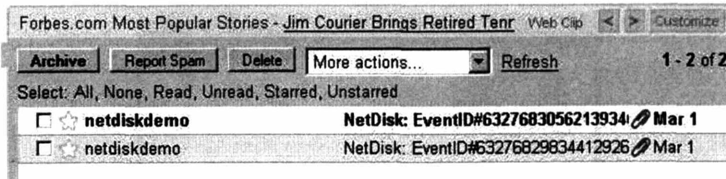


Figure 6-1: Gmail account used for Universal Object Storage.

to an email account. The Simple Mail Transfer Protocol (SMTP [2]) can be used to send an email to an email account. The data can be serialized and packed as an attachment with the email. We simply put the object reference string as the subject of the email message and leave the text content of the email blank. To retrieve the data object, we use the GmailAPI [5] to read information from the Gmail account, including reading an individual email message. The data object previously saved is unwrapped from the attachment of the email message.

In particular, when the Write() function is called by a remote data sender, the GmailStorage serializes the object, encodes it into a Base64 character block as an attachment and adds headers to comply with the standard of Internet email. It then connects to Gmail SMTP server [6] and delivers the email message to the Gmail account. Figure 6-1 shows the Gmail account with samples of those messages. Later when the Read() function is called, the GmailStorage calls a corresponding function of GmailAPI to search and fetch the email message with the subject identical to the requested object reference ID. It then decodes the attachment, de-serializes the object and returns the object to the caller.

Due to the limitation of the size of the email attachment allowed by Gmail, the GetSingleObjectLimit() is overridden by the GmailStorage and returns 1048576 [7].

[5]GmailAPI is an opensource package used to access a Gmail account through Gmail's XML interface. The project is no longer maintained.

[6]This is done by query a DNS server for the MX record of Gmail.com [27].

[7]1 megabyte.

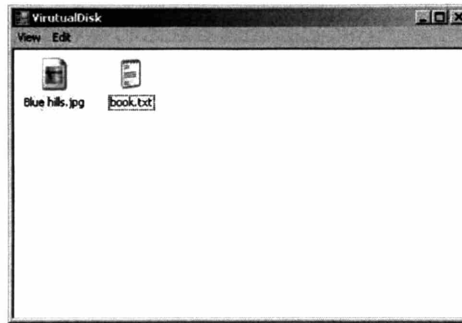


Figure 6-2: The virtual file browser.

6.4 A Virtual File Browser for the Universal Object Storage

Although the implementation and backend storage media of this example is very different from the previous one, they have the same interface and can be used for the same task. Figure 6-2 shows the appearance of a virtual file browser very similar to the Windows file browser [8]. It can connect to any Universal Object Storage using a hostname and allows drag-and-drop style operations.

In this chapter, we demonstrated how we could use the Universal Object Storage interface to build a proxy between the data users and the underlying storage media. In particular, although I do not see a potential future of the Gmail storage application, it is a good demonstration for the versatility of the Realm system.

[8] Although it looks very similar to the file explorer, its functions are restricted. For example, file folders are not supported.

Chapter 7

The Messenger Application

The Messenger programming interface offers a channel-based collaboration framework for conferencing-like systems. As was explained in Chapter 3, the Messenger interface closely imitates the scenario of an ordinary meeting. Building up an Instant Messenger or a conferencing software [1] based on it is very straightforward. In this chapter, we present an application used with Radio Frequency Identification (RFID) in a supply chain.

7.1 Problem Introduction

RFID is an automatic identification method, relying on storing and remotely retrieving data using devices called RFID tags. An RFID tag is a small object that can be attached to a product and has a micro circuit as well as an antenna to enable it to receive and respond to electromagnetic queries from an RFID transceiver. An organization called GS1 [2] operates the joint venture EPCglobal [3], which is working on international standards for the use of RFID in the identification of any item in the supply chain for any company in the world. Today, as universal RFID tagging of individual products becomes commercially viable at very large volumes, the lowest

[1] Since the Realm Framework has not been able to support streamed data, video/audio conferencing is not applicable.

[2] http://www.gs1.org

[3] http://www.epcglobalic.org

cost tags available on the market are approximately 7.2 cents each [4].

The MIT Auto-ID Lab is actively involved in the research topics related to RFID applications. One interesting topic is how to coordinate one or more supply chains, allowing the users to both publish and subscribe to events and to launch queries against the aggregated data. The requirements for a supply chain management system are listed below.

1. Individual supply chain management must be relatively self-maintained and must offer a safe method to access the chain from outside.

2. Each supply chain has multiple event channels published so that the participants in the chain can subscribe to one or more of them and receive the events they are interested in.

3. A security model should be provided to prevent the system from becoming an ad-hoc network.

4. The system should support a variety of data types passed as events.

5. Supply chains should be able to be located on the Internet by simple means.

We are able to develop such a supply chain management system with the Messenger interface provided by the Realm Framework. For the purpose of example, we do not develop sophisticated functionality such as a data warehouse in this proof-of-concept application.

7.2 A Simple Supply Chain Communication System

In this sample application, there are multiple roles: supply chain manager, manufacturer, shipping service company, retailer and lost-and-found station (Figure 7-1).

[4]This information is from an online news at http://www.smartcodecorp.com/newsroom/05-10-05.asp.

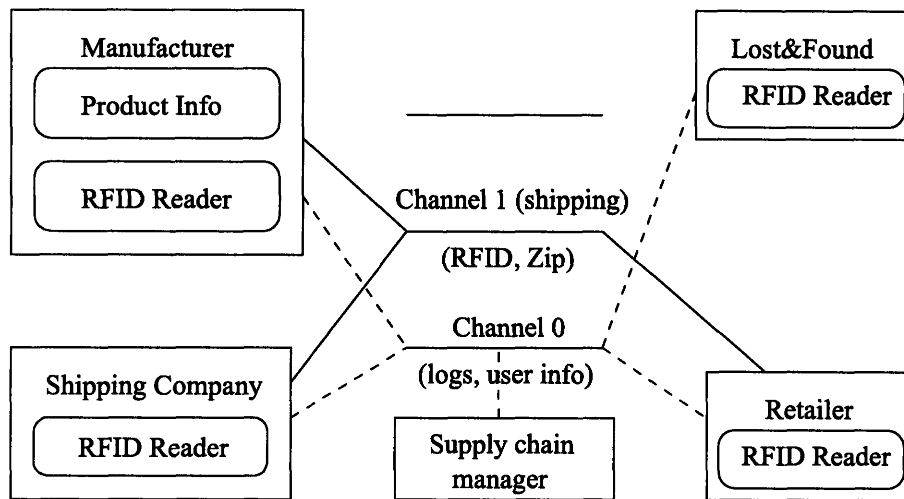


Figure 7-1: The parties in a supply chain.

The supply chain manager initializes the communication system and maintains the communication channels. The manager can be a supply chain management service provider or simply a company involved in the chain. The manufacturer is where the product information originates. The shipping service company carries the product to different locations, in our example, identified by a zip code. The retailer is the receiver of the product while the lost-and-found station offers a service to identify missing products.

The supply chain manager starts the Realm job and registers the job on the Realm Server so that the job can be located and accessible from anywhere. All other participants [5] need to log into the Realm and join the job [6]. They are monitored by the supply chain manager. All public messages, such as log-in and log-out notification, are passed through channel 0 headed by the supply chain manager. Other private channels can be acquired by other participants who then have to define a group of eligible event receivers. A participant can also join a channel if it is invited. In this implementation, invitations are always honored. Once a private channel has been established, all participants [7] are able to broadcast or receive private events

[5]Here the participants refer to the other Worker processes.

[6]A job here is used in the context of distributed programming. It has the same meaning as a supply chain.

[7]Or in another word, the subscribers.

particular to the channel. For example, the shipping company can head a channel named "Shipping" and invite the manufacturer and retailer as participants. Only events related to shipping will go through the "Shipping" channel to notify the other parties. This guarantees the privacy of the message and the efficiency of the message processing.

Due to the nature of the Realm system's programming model, the event can be any serializable object. In this example, there are "GeneralMessage", "Order", "Receipt", "ProductInfo" and "RFIDReading" classes. In particular, the RFIDReading object is a blob containing XML information complying with the EPCglobal standards.

As a typical case, the retailer first issues an order by sending an "Order" object through the "Ordering" channel. The manufacturer later broadcasts through the "Shipping" channel with a ProductInfo object wrapping up the product information. The product then goes through the check points of the shipping company. At each check point, the RFIDReading object goes through the "Shipping" channel again so that the manufacturer and the retailer can monitor the shipping procedure continually. A Receipt object is sent by the retailer to notify the manufacturer that delivery was successful. The lost-and-found station sends an RFIDReading object through the public channel upon identification of a missing package. Figure 7-2 is a screen-shot of the actual supply chain demo.

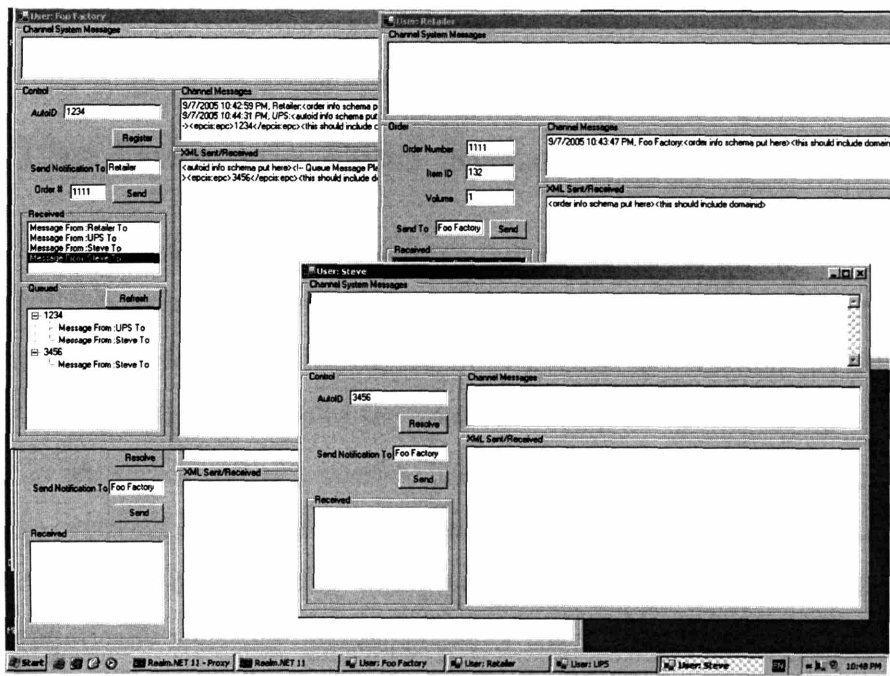


Figure 7-2: Supply chain emulator interfaces.

Chapter 8

Supporting Components

8.1 The Dynamic DNS System

Locating network resources usually follows one of two main patterns: label-based naming (LBN) or description-based naming (DBN). LBN systems affix a label to an object and use it to locate and access the object. DBN systems, on the other hand, use a set of attribute-value tuples to describe an object. Even though it provides flexibility in answering resource queries, it comes at the cost of additional overhead. Most of the overhead is associated with maintaining databases of the attribute-value tuples and resolving queries using values within these databases. The overhead increases dramatically with the increasing size of the network. [11]. The DNS tree is a typical and popular LBN system, while the grid computing community tends to use DBN to locate resources. In the Realm Framework, service location relies on the DNS system, not only because the hostname is considered sufficient to describe a resource in the Realm Framework context, but also because the DNS system has so far been so popular and familiar that even a non-programmer can understand the idea–each hostname universally identifies a resource object. Using online resource objects provided by Realm systems is as simple as surfing the Internet. For example, "svc1.iesl.mit.edu" is a resource object location mentioned before (Figure 8-1).

A resource is often floating because service workers are volunteer-based, and the resource maintainers have the freedom to choose which Realm they want to join. For

this reason, the IP address matching the hostname of the resource may change frequently. This raises a difficulty for registering the service on UDDI because typically a fixed location is desired.

This problem can be solved by using the Dynamic Domain Name system (DDNS). By setting the Time To Live (TTL) property of the hostname record to zero, the hostname record will not be cached on mid-way DNS's and thus becomes a *dynamic host* [27]. Although real time updating features are available for some popular DNS software, such as BIND (ref: bind), a Web service enabled, lightweight DDNS software was developed from the ground up to provide seamless connectivity with other components of the Realm framework. Resource registration (adding updating a hostname record) and server management are done through a set of Web services. As a sample scenario, suppose a resource with the name "foosvc.iesl.mit.edu" is registered under the "iesl.mit.edu" mother resource tree through the DDNS Web services. Since the IP address of the Realm Server holding the resource object "foosvc.iesl.mit.edu" is "18.58.0.199", the newly updated record in the DDNS for "foosvc.iesl.mit.edu" now points to "18.58.0.199". This record is then stored in a database server. Next time when any network client wants to access the resource object and call the service "foosvc.iesl.mit.edu", this hostname string will be resolved to the IP address of the Realm Server–"18.58.0.199". If the client software is Realm-aware, it will usually call the "CallService" method at the "http://18.58.0.199/Service" Web address.

Although not mentioned explicitly, this service identification and location scheme has been used in some of the applications described in the previous chapters. For example, the Movie Maker program relies on the hostname of the storage service so that it can save the generated frame images on the Universal Object Storage node; the Lattice-Boltzmann simulator exposes intermediate data as Globals and allows a standalone graphic interface to locate them through a hostname and disply them graphically; the supply chain simulator identifies each supply chain as a unique hostname so that participants of the supply chain can join the chain from anywhere.

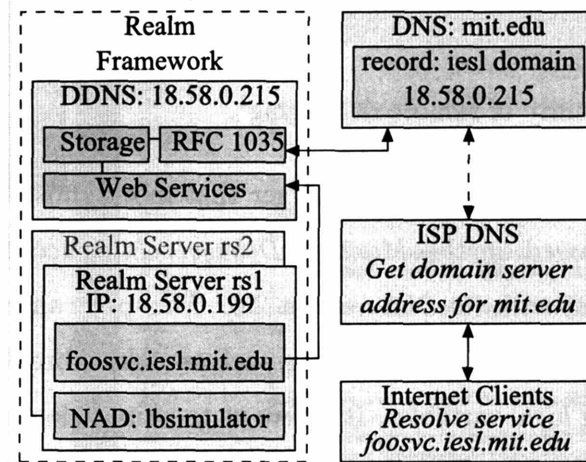


Figure 8-1: The Realm Dynamic DNS System.

8.2 The Realm Debugger

Without a debugger, developing software is a painful process. Debugging a parallel computing application is even harder because we have to take care all the computing processes at the same time. The Microsoft .Net Framework offers a powerful debugger called "cordbg". The cordbg is designed for an ordinary single machine programs, however. It provides some basic debugging capability and allows us to find out how the Worker process is running on a local computer. In this case the cordbg is debugging the WorkerI.exe, and it requires some tricks to distinguish the Worker processes hosted by the WorkerI.exe program. This is a time-consuming job and requires technical proficiency. A good distributed system should provide a debugger that:

1. offers a rich set of capabilities comparable to those offered by a non-distributed debugger;

2. hides the distributed framework when necessary so that the application programmer can just focus on the application itself;

3. identifies individual processes and provides the information pertaining to the underlying distributed system, such as process ID, NAD and Seed type for the Realm system;

4. offers a graphical debugger interface to increase productivity;

5. offers the capability of remote debugging[1].

The Realm system comes with a debugger that meets most of the requirements listed above. It is based on the *Managed Debugger Interface* published as a code sample [2] by Microsoft with its .Net version 2.0. This programming interface offers most of the functions provided by the cordbg, such as making break points, responding to breaks and getting local variables. It allows us to write a debugger using C#. The Realm debugger merges the capability of the Managed Debugger Interface and the specifications of the Realm system with a graphical interface. When we launch the debugger interface, it will first search the WorkerI.exe process on the local machine. If the WorkerI.exe is up and running, the debugger asks the user whether it should attach to the WorkerI.exe immediately. Attaching to it means starting a debugging and all the Worker processes managed by the WorkerI.exe process are paused. The Realm debugger then analyzes the break points of all the threads, and identifies the ones belonging to the Worker processes but not the Realm system process. The debugger then proceeds to find out the process ID of these Worker processes and the distributed jobs they belong to. Finally, the debugger loads the source file for each job and allows the user to debug them. The job control can be fine-tuned to each individual Worker process, for example, pausing the processing of one Worker but allowing the other Workers to proceed as usual. The debugger GUI looks like the Visual Studio interface and should be familiar to most C# programmers (Figure 8-2).

8.3 Limitations

The Realm debugger provides a set of functions that helps the programmer in most debugging scenarios. However, there are still many features that are not yet included. The Realm debugger can not do remote debugging currently, which means only the

[1]Not yet implemented.

[2]You can get this on MSDN by searching "CLR Managed Debugger (mdbg) Sample".

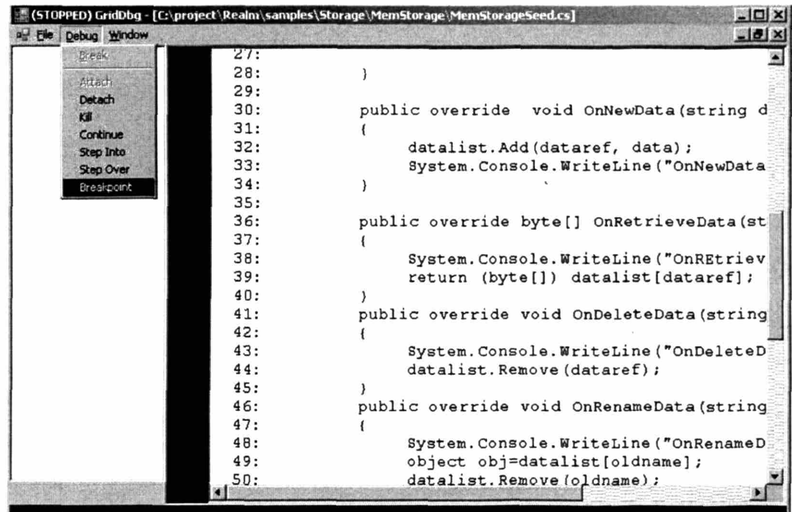


Figure 8-2: The Realm Debugger interface.

local Worker processes are visible to the debugger. For this reason, the Realm debugger is not a true distributed debugger. Besides, for each job, the Realm system usually runs the same DLL file. This means the Worker processes under the same job usually use identical source code files. The break points are set according to the location inside these files. The current debugger can not set different break points for different Workers in this case. For example, if Worker process 0 has a break point at line 100 of source file A.cs, this also means all the other Workers being debugged will have the same break point at line 100 of file A.cs. Another limitation is we have to point out the location of the debugger symbol files [3]for the Seed DLL, which can be inconvenient.

[3]The "program debug database" file, or the pdb file, is generated by the compiler.

Chapter 9

Conclusion

9.1 Summary

This thesis proposes the Realm Framework as a new concept for the sharing of Internet resources. The implementation is based on the Microsoft .Net Framework and the C# programming language.

We began in Chapter 2 with a brief examination of the previous resource sharing frameworks, including typical parallel computing systems focusing on sharing computing power and the more recent, general-purpose resource sharing ideas, such as Grid Computing. In Chapter 2, we identified the weaknesses of the previous projects and the potential use cases employing current technology. The conclusion of the review is that an alternative to these systems is truly necessary.

Chapter 3 started with an explanation for the choice of the .Net Framework and C# as the underlying platform for the Realm system. In this chapter, we showed that our design and implementation carefully considered the accessibility, programmability, reliability and efficiency. We have achieved the goal of the study.

Chapter 4, 5, 6 and 7 focused on applications using the Ream Framework. The Realm system was proved to work well under the same situation as SETI@Home and MPI. Additionally, the study of the performance showed that different network configurations, especially when we use encryption with .Net Remoting, influence the runtime performance in a predictable way. With additional support for storage and

messenger-style applications, we conclude that the Realm Framework is versatile and meets the requirement as a general-purpose resource sharing framework.

Finally in Chapter 8, we discussed two miscellaneous issues, namely the method by which we identify and locate the resources, and the Realm debugger as a handy tool for application developers. Although they are not major components of the Realm Framework itself, they make development under and usage of the Realm Framework much easier. We further conclude that the Realm Framework is a feature-rich platform sufficiently to be an alternative to the other systems.

In summary, this thesis contains rich information which paves the road for future research.

9.2 Contributions

The major contributions of this thesis include:

1. **The Realm concept and its implementation.** We realized the limitation of the previous systems and have presented the design and various aspects of the implementation of the Realm Framework. It is an initial work that is significantly different from other systems.

2. **The programming interfaces for various real-world problems.** We have developed a layered programming-interface hierarchy so that application programmers can choose the interfaces most suitable for the particular task.

3. **A rich set of applications that have already been developed based on the Realm Framework.** To serve as a start-point and roadmap to develop quality applications using the Realm Framework, many demonstration programs, some of which are serious engineering applications, were presented in this thesis.

4. **Approaches to the problem of security and robustness under the Internet environment.** Encryption, authentication and runtime security were

carefully studied. Fault-tolerance was also a topic of the research. Improvements in the initial design and implementation have been done to enable a safe Realm over the Internet.

5. **Research about the performance of the Realm Framework.** We presented the results of performance comparisons for various application scenarios. The research not only proves the usability of the Realm system, but also suggests the optimal configuration for different purposes.

6. **In-depth research about parallelization for DEM.** As an additional contribution, this thesis proposed a domain partitioning method for DEM. Our experiments showed that it was efficient and easy to implement with the Realm Framework.

In summary, this thesis is an fundamental step towards a new distributed resource sharing methodology. We hope that it can help further research in this area, and the developed system can serve as the startpoint of a more powerful software package.

9.3 Future Work

Although the Realm Framework even at its current stage allows us to develop distributed programs and use Internet resources in many cases, it is far from a complete software system. We have pointed out some of the current system's limitations in the thesis. In light of these limitations, I suggest the following topics to guide future research:

1. **WSRF Compatibility.** WSRF stands for Web Services Notification and Web Services Resources Framework [15]. WSRF is the most recent open standard designed to merge grid and the Web technologies in terms of today's Web service standards. Complying with WSRF would enable the Realm Framework to communicate with other distributed systems.[1] We have not explored this issue

[1] Currently the WSRF.Net package developed in University of Virginia Grid Computing Group allows Windows programmers to adapt their Web services to be WSRF-compatible. See their Web site: http://www.cs.virginia.edu/ gsw2c/wsrf.net.html.

yet.

2. **Mobile device support.** In Chapter 2 we mentioned that the mobile device support was once in the thesis plan but finally left undone due to the lack of development software support for these devices. As mobile devices are getting more and more popular, this research may be attractive in the future.

3. **More abstract programming models for more fine-grained application scenarios.** For example, the current communication-intensive interface is the general platform to develop parallel computing applications.The programmer, however, may prefer a more specific model, say, a model that wraps up the widely used ghost area and offers a set of abstract boundary classes that can take care of various boundary conditions without writing code from scratch. Future research can build a higher degree of abstractions based on the current models to directly support a variety of applications.

4. **Compression sink.** We currently have the authentication and encryption Remoting sinks. As we have pointed out in Chapter 5, the communication cost of our current software is a non-trivial issue. Compressing the messages may be a solution to reduce the communication cost. The trade off between the transmission gain from data compression and the over-head introduced must be studied.

5. **Failure handling and fault-tolerance for applications other than simple master-slave style ones.** We have discussed a possible scheduler for the master-slave model in Chapter 3 to handle failure and faults. Relative studies for the other programming interfaces have not been performed. Further research might start from the easiest case–the Universal Object Storage interface, and move toward the most difficult–the communication-intensive parallel computing interface.

6. **Coupling the Lattice-Boltzmann method with DEM.** We have studied the Lattice-Boltzmann method and DEM in Chapter 5. Putting them together

to simulate fluid flow with solid objects is another challenging topic.

7. **Method to find the optimal reference axis for distributed DEM.** The new partitioning model we used in the DEM simulator is not fully automatic, because the user still need to find an optimal reference axis. If the reference axis can be found automatically, the efficiency of using the simulator will be greatly improved.

8. **Distributed storage.** The current Universal Object Storage model does not support true distributed storage as GridFTP does. This is a challenging task because we should not only consider the implementation, but also the programming model offered to application programmers.

9. **Remote debugging.** The current Realm debugger is not a truly "distributed" debugger because it does not support remote debugging. Remote debugging is a necessary feature for a distributed system.

9.4 Final Words

It is my hope that this thesis is truly helpful for colleagues within the same research area who are interested in working on a distributed system that gives hope to the vast majority of the computer users. The Realm idea makes some initial steps toward this goal. We would like to see a fruitful future for the Realm concept.

Bibliography

[1] *OGSA (OpenGridServices Architecture, 2002).* online at http://www.globus.org/ogsa/.

[2] *Simple Mail Transfer Protocol.* Request for Comments: 821.

[3] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide.* Society for Industrial and Applied Mathematics, Philadelphia, PA, 3 edition, 1999.

[4] M. Baker, B. Carpenter, S. Ko, and X. Li. mpijava: A java interface to mpi. In *First UK Workshop on Java for High Performance Network Computing*, Eoropar 98. 1998.

[5] G. Buda, D. Choi, R.F. Graveman, and C. Kubic. Security standards for the global information grid. *Communications for Network-Centric Operations: Creating the Information Force*, 1:617–621, 2001.

[6] H. Chen, S. Chen, and W.H. Matthaeus. Recovery of the navier-stokes equation using a lattice-gas boltzmann method. *Physics Review*, A(45):5539–5542, 1992.

[7] S. Chen and G.D. Doolen. Lattice boltzmann method for fluid flows. *Annual Review of Fluid Mechanics*, 30:329–364, 1998.

[8] P.A. Cundal and O.D.L. Strack. A distinct element model for granular assemblies. *Geotechnique*, 29(1):47–65, 1979.

[9] I.G. Currie. *Fundamental Mechanics of Fluids.* McGraw-Hill, 1974.

[10] P. Dasgupta, Z. Kedem, and M. Rabin. Parallel processing on networks of workstations: A fault-tolerant high performance approach. In *Proceedings of 15th IEEE International Conference on Distributed Computing Systems.*

[11] P. Dinda and B. Plale. *A unified relational approach to grid information services*, Feb 2001. Grid Forum Informational Draft GWD-GIS-012-1.

[12] Uriel Feige and Robert Krauthgamer. A polylogarithmic approximation of the minimum bisection. *SIAM J. Comput.*, 31(4):1090–1118, 2002.

[13] The MPI Forum. Mpi: A message passing interface. In *Proceedings of Supercomputing '93*, page 883, Portland, Oregon, 1993.

[14] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman, Jan 1979.

[15] Globus. *WSRF (Web Services Notification and Web Services Resources Framework, 2004).* online at http://www-106.ibm.com/developerworks/webservices/library/ws-resource/.

[16] S.W. Golomb. *Graph Theory and Computing*, chapter How to Number a Graph, pages 23–37. Academic Press, 1972.

[17] B. Hayes. Collective wisdom. *American Scientist*, 86(2):118–122, Mar-April 1998.

[18] S. Hou, Q. Zou, S. Chen, G. Doolen, and A.C. Cogley. Simulation of cavity flow by the lattice boltzmann method. *Journal of Computational Physics*, 118:329–347, 1995.

[19] R. Housley, W. Ford, W. Polk, and D. Solo. *Internet X.509 Public Key Infrastructure Certificate and CRL Profile.* Network Working Group, 1999. Request for Comments: 2439.

[20] J.K. Huggins. *Specification and Validation Methods*, chapter Kermit: Specification and Verification. Oxford University Press, 1995.

[21] S. Johnson. *Resolution of Grain Scale interactions using the Discrete element method.* Phd thesis, Massachusetts Institute of Technology, 2005.

[22] N.T. Karohis, B. Toonen, and I. Foster. Mpich-g2: A grid-enabled implementation of the message passing interface, Nov 2002. published online by Globus (http://www.globus.org.

[23] X. Lin and J.R. Williams. A grid computing architecture for applications in discrete computational mechanics. In *7th US Congress on Computational Mechanics*, Albuquerque, NM, Jul 2003.

[24] N.A. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1 edition, 1997.

[25] Microsoft Corporation. *SMB File Sharing Protocol Version 6.0p*, Jan 1996. online at ftp://ftp.microsoft.com/developr/drg/CIFS/smbpub.zip.

[26] Microsoft Corporation. *Windows Compute Cluster Server 2003 Beta 2 Reviewers Guide*, Nov 2005.

[27] P. Mockapetris. *DOMAIN NAMES - IMPLEMENTATION AND SPECIFICATION.* Network Working Group. Request for Comments: 1035.

[28] Y. Qian, S. Succi, , and S. Orsazg. Recent advances in lattice boltzmann computing. In D. Stauffer, editor, *Annual Reviews of Computational Physics III*, pages 195–242. World Scientific, New Jersey, 1995.

[29] Y.H. Qian, D. D'Humieres, and P. Lallemand. Lattice bgk models for navier-stokes equations. *Europhysics Letter*, 11:479–484, 1992.

[30] L.F.G. Sarmenta. Bayanihan: Web-based volunteer computing using java. In *Proceedings of the 2nd International conference on World-Wide Computing and its Applications*, Tsukuba, Japan, Mar 1998.

[31] L.F.G. Sarmenta. *Volunteer Computing.* Phd thesis, Massachusetts Institute of Technology, Jun 2001.

[32] N. Satofuka and T. Nishioka. Parallelizaiont of lattice boltzmann method for incompressible flow computations. *Computational Mechanics*, pages 164–171, 1999.

[33] S. Shepler. *NFS Version 4 Design Considerations.* Network Working Group. Request for Comments: 2624.

[34] A. J. van der Steen and J. J. Dongarra. Overview of recent supercomputers. Technical report, TOP500.Org, 2003. TOP500 report.

[35] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. Extensible security architecture for java. In *Proceedings of the 16th Symposium on Operating System Principles*, pages 116–128, St. Malo, France.

[36] J.R. Williams and X. Lin. A grid computing architecture for applications in distributed computation. In *2nd International PFC Symposium, Numerical Modeling in Micromechanics via Particle Methods*, Kyoto, Japan, Oct 2004.

[37] J.R. Williams, E. Perkins, and B. Cook. A contact algorithm for partitioning n arbitrary sized objects. *Engineering Computations: Internatioal Journal for Computer-Aided Engineering*, 21(2-3):235–248, 2004.

[38] J.R. Williams X. Lin. A parallel computing framework for computer cluster. In *Proceedings of the 16th IASTED International Conference on Parallel And Distributed Computing And Systems*, Cambridge, MA, Nov 2004.

[39] L.-J. Zhang and M. Jeckle. Convergence of web services and grid computing. *International Journal of Web services Research*, pages "i–iv", "Jul-Sep" 2004.