APPROVAL PAGE FOR GRADUATE THESIS OR PROJECT

SUBMITTED IN PARTIAL FULFILLMENT OF REQUIREMENTS FOR
DEGREE OF MASTER OF SCIENCE AT CALIFORNIA STATE
UNIVERSITY, LOS ANGELES BY

Kamyar Miremadi
Candidate
Computer Science
Field of Concentration

TITLE: Building a Peer to Peer Environment for Message Passing Interface by Utilizing Reflection

APPROVED:

Dr. Behzad Parviz__________________________
Faculty Member Signature

Dr. Raj S. Pamula__________________________
Faculty Member Signature

Dr. Raj S. Pamula__________________________
Department Chairperson Signature

DATE: 08/21/06
BUILDING A PEER TO PEER ENVIRONMENT
FOR MESSAGE PASSING INTERFACE BY UTILIZING REFLECTION

A Thesis
Presented to
The Faculty of the Department of Computer Science
California State University, Los Angeles

In Partial Fulfillment
Of the Requirements for the Degree
Master of Science

By
Kamyar Miremadi
August 2006
ACKNOWLEDGEMENT

They say that it takes a whole village to grow a child. What reaches the page is the product of many minds and hands. Here is the place to thank those without whose generous concerns; this project would have remained a series of half-baked one-liners.

I would like to express my sincere thanks and appreciation to all those student colleagues and friends whose helpful comments, support, and encouragement were invaluable to me in the preparation of this proposal and leading it to an end, though still many points need to be mentioned and added to the results in future. Particular thank is due to Professor Behzad Parviz for his direction, assistance, and guidance. His recommendations and suggestions have been invaluable for the project and for the software improvement. He was kind enough to carefully read and review all of the manuscript in its various stages. However, I take the responsibility of any pitfalls observed in the presentation of ideas.

I also wish to thank Dr. Raj Pamula, Dr. Valentino Crespi, Dr. Jiang Gue, Dr. Cheng Yu Sun, Mr. Edmond Gean, Dr. Russell Abbott and other faculties who have all taught me different aspects of computer science.
ABSTRACT

Building A Peer To Peer Environment For Message Passing Interface

Utilizing Reflection

By

Kamyar Miremadi

Nowadays many projects based on peer to peer computing (one form of grid computing) are being conducted world widely. Grid computing (or the use of a computational grid) is a system which enables the users apply the resources of many computers in a single network endeavoring to find solutions to a single problem at the same time usually to a scientific or technical problem that requires a great number of computer processing cycles or an access to large amounts of data.

MPI is a standard protocol that is widely utilized in many of the parallel and distributed applications. In other words, MPI is a communicating programming model for general purposes, particularly in parallel and distributed applications. Our solution benefits from .Net Web Service Technology for Communication Services and Reflection for Execution Model. Reflection is the ability of a program to modify its own behavior in the course of its evaluation.
Our approach implements class libraries in .NET to make them available to the programmers and enable them to write a quick and easy to use parallel and distributed program as well as internal classes that are used in our communication and execution model.
# TABLE OF CONTENTS

Acknowledgement ...................................................................................... iii

Abstract ..................................................................................................... iv

List Of Figures .......................................................................................... viii

Chapter 1 Introduction .............................................................................. 1

Chapter 2 System Architecture Model ..................................................... 4
  2.1 Utilizing of .Net Web Services .......................................................... 4
  2.2 Reflection in .NET ............................................................................. 6
  2.3 Execution Model ............................................................................... 8

Chapter 3 System Features and their Implementation ............................. 14
  3.1 Implementation of MPI Protocol ...................................................... 14
  3.2 Implementation of Remote Function Calls ...................................... 17
  3.3 Sample Programs ........................................................................... 19

Chapter 4 Advanced Features ................................................................. 22
  4.1 Distributed Hash Table (DHT) Algorithms ...................................... 23
  4.2 Chord Algorithm .............................................................................. 24
    4.2.1 Joining and Leaving of Nodes To/From Network ....................... 28
    4.2.2 Get Successor Function ........................................................... 31
    4.2.3 Refreshing of Finger Table ....................................................... 34
  4.3 Check-Pointing Feature .................................................................. 35

Chapter 5 Conclusion ................................................................................. 39
References ......................................................................................................................... 40

Appendices ......................................................................................................................... 43

  Appendix A: Server and Client Services Class Diagrams (Final Design) ................................. 43

  Appendix B: Business Classes Diagrams ............................................................................. 45

  Appendix C: Graphical User Interfaces ................................................................................. 50
LIST OF FIGURES

Figure 1: Server Service Class Diagram ....................................................9
Figure 2: Client Service Class Diagram ....................................................10
Figure 3: Class Diagram of Client Collaboration Objects .........................11
Figure 4: Nodes Placement in CHORD Algorithm .......................................25
Figure 5: Nodes and their Finger Table in CHORD ...................................26
Figure 6: Sequence Diagram for Joining of a Node to the Network ..........29
Figure 7: Sequence Diagram for Leaving of a Node from the Network ....30
Figure 8: Server Service Class ...............................................................43
Figure 9: Client Service Methods and Attributes .......................................44
Figure 10: Business Classes Class Diagram ..............................................45
Figure 11: ProgramController Class Methods and Attributes .................45
Figure 12: DistributedApplication Class Methods and Attributes ..........46
Figure 13: Program Class Methods and Attributes ....................................47
Figure 14: Group Class Methods and Attributes .......................................47
Figure 15: Peer Class Attributes .............................................................48
Figure 16: Instance Class Methods and Attributes ...................................48
Figure 17: OutPipe Class Methods And Attributes ..................................48
Figure 18: InPipe Class Methods And attributes ....................................49
Figure 19: Management Page for Monitoring Server Service ..................50
Figure 20: Displaying of List of Items in Finger Table .............................50
Figure 21: User Interface for Managing Client Service, Loading and Executing of Programs .................................................................51
Chapter 1

Introduction

Nowadays many projects based on peer to peer computing (one form of grid computing) are being conducted world widely. Grid computing (or the use of a computational grid) is a system which enables the users apply the resources of many computers in a single network endeavoring to find solutions to a single problem at the same time -usually to a scientific or technical problem that requires a great number of computer processing cycles or an access to large amounts of data.

Those who need to have access to large amounts of data can also benefit from the model as it has been developed in this project. Projects like SETI@HOME [13] use the similar approach to do a large number of computations by using volunteer computers through the Internet System.

Almost all approaches address the need for sharing computing cycles. Seti@Home , Folding@Home [17], Einstein@Home [18], ExtremeWeb [19] and many other similar approaches focus on using idle CPU power of the volunteers around the world for the purpose of conducting science researches. On the other hand, many other researches such as G2:P2P [4], Classic G2 [8], G2 Remoting [7] and

However, it needs to be mentioned that some of these approaches are limited to a single functionality and do not provide multi-purpose environments for any type of application. MPI [14] is a communicating programming model for general purposes (parallel and distributed applications) and is widely used. In our implementation, we benefit from a similar MPI approach in a peer to peer environment.

In our approach, programmers can share their processors and network resources voluntarily for a parallel faster execution. To achieve this purpose, the programmers need to generalize and to override a class inside their class libraries. This class provides them an environment within which they can get access to the main MPI procedures and functions that are needed for a parallel programming. The program is thus encoded to the client-side program service through its source code or by a compiled DLL .Net Assembly. Enjoying the two options aforementioned, the program is compiled dynamically and will be dispatched to the required number of nodes for parallel executions. The program is then executed in the peers. We enjoy .NET
XML Web services [15, 16] to provide network communication and use run-time Reflection [20] for execution of the program dynamically.

Reflection is the ability of a program to modify its own behavior in the course of its evaluation. It was first studied in logic and philosophy and then arose in AI and then it linked itself to programming language paradigm as a computational reflection. Two types of Reflection have been enlisted and categorized: Behavioral Reflection, which is the ability of the program to alter its own meaning by manipulating itself and its evaluator. Linguistic Reflection, on the other hand, refers to the ability of program to alter its data structures and to add new coding. In modern programming language and environment like the ones such as Java and .NET, it is possible to use Linguistic Reflection too. Linguistic Reflection furnishes a programming system the ability to generate new program fragments and to incorporate them into the on-going computation.
Chapter 2
System Architecture Model

Unlike Java, .Net framework doesn't have APIs for peer to peer networking. Java benefits from open source JXTA [10, 11] Technology (Developed by Sun Microsystems) that allows any device connected to the network to exchange messages. This Technology is a P2P framework that allows peers to communicate one another on a virtual overlay network even if some of the peers are behind a firewall or are using a different network transportation layer.

Due to certain shortcomings, providing peer to peer functionality in .NET requires more adequate and collaborative programming efforts. However, it needs to be mentioned that .Net Framework is rich of many client-server communication technologies such as .Net Remote Services and Web Services such that enable the users to get accesses to the intended goals easily.

2.1 Utilizing of .Net Web Services

In our model, we have utilized web services. Web services have some advantages over the Remote services and seem to be more efficient: The merits are briefly listed below:
• Web services are language and platform independent. Due to the fact that Web services use WSDL, which is an XML format-oriented technology to describe the service and SOAP as an exchanging message protocol, and very promising, it is plausible and quite desirable to define and to consume web services in different languages.

• By Utilizing HTTP protocol, web services can work through the firewalls.

• Web Services are loosely coupled so they can provide better reuse and distributed integrations of applications.

• In contrast to remote services, it is easier and more cost-effective to find a host for a Web service.

• .Net comes with free Webdev.Webserver application and can easily be utilized in our system. Moreover, other free web servers such as UltiDev Cassini can be used. Microsoft Information Services Manager (IIS)\(^1\) can also be used as well. It reduces lots of coding needed with the similar approaches.

Beside these advantages thus listed above, Web services face few challenges. Web services are connectionless and stateless. Web services provide us only a mechanism for Remote Procedure Call

\(^1\) IIS is built in Microsoft Windows XP Professional and Window 2000 and 2003 Server
(RPC) through transferring requests and, in return, through receiving responses. In other words, it seems to be rather improbable to manage an object in a server from a client without providing some persistent methods. In our implementation, we used singleton patterns [9]. This approach requires exchanging of GUID Object IDs between the server and its clients.

2.2 Reflection in .NET

Reflection is the ability of a program to modify its behavior at runtime. Reflection makes it possible to discover class information at runtime. Through using Reflection, the possibility of loading an assembly in runtime, creating objects from classes and even calling the methods or getting access to the fields and properties of those objects is fortified. In .NET Framework, `System.Reflection` and `System.Reflection.Emit` namespaces provide APIs needed to handle these kinds of tasks. The former provides APIs to work with a compiled assembly and the latter contains classes that allow a compiler or any other tools to emit a Meta data and Microsoft Intermediate Language (MSIL) into an assembly.

Furthermore, `Microsoft.CSharp` and `Microsoft.VisualBasic` namespaces have some other classes that allow dynamic compilation of the code to be traced from C# and VB.NET language.
The code below shows how we benefit from Reflection to load an assembly and create and give reference to an object at runtime.

```csharp
public DistributedApplicationClass CreateP2PObject(byte[] AssemblyCode)
{
    Assembly Assm = Assembly.Load(AssemblyCode);
    Type DistributedApplicationType = new DistributedApplicationClass().GetType();
    foreach (Type typ in Assm.GetTypes())
    {
        if (typ.IsSubclassOf(DistributedApplicationType))
            return (DistributedApplicationClass)Activator.CreateInstance(typ);
        return null;
    }
}
```

In our programming model, each application must inherit from `Distributed Application` Class. We will discuss the Class Architecture in more detail as follows. In the code above, an assembly is loaded and then it checks all defined classes in the assembly. If any of the classes in the assembly is of the type `Distributed Application` Class, it can create an object and return it as the result. It is also possible to call a method at runtime.

```csharp
public void RunMethod(int fromRank)
{
    FunctionSignatureClass FuncSig = (FunctionSignatureClass)Receive(fromRank);
    Type mytype = this.GetType();
    MethodInfo Mymethod = mytype.GetMethod(FuncSig.MethodName);
    object result = Mymethod.Invoke(this, FuncSig.Parameters);
    Send(fromRank, result);
}
```
The above code implements part of the remote method calling between peers. It reads the value received from a peer and by using Method Info Class, looks for the method with the given name, invokes the method on current object and sends the result back to the Run Method caller.

2.3 Execution Model

In our initial model, we built the server centric system. The server is nothing more than a .Net web service that manages connecting and disconnecting of peers to the overlay network and maintains some other information about peers' current tasks such as execution information (Who executes the application, whatever the group is or what ranks they possess). In the next step, these server services extend to support de-centralized querying of their connected node information. We will address this new feature and its implementation in Chapter Four. The following figure shows the Methods that were implemented in the server centric web service.
Once an application is loaded into the local client, it gets a Unique GUID ID. Each application may be executed several times for a certain number of peers. In this case, the client utility which itself is another web service on the local machine, interacts with the server web service to request URL addresses of the peers that are needed for a parallel execution. In the next step, the local client creates another GUID ID for this execution group, dispatches application assembly and execution information like ProgramOID, GroupOID and the rank of the running instance to the colleague peers. The client web service on each one of the peers then loads the assembly, sets its execution info into the running instance object, and then executes it.
When an assembly is loaded, the client looks for the class that inherits from `DistributedApplication` class. Having found the class, it initializes an object from that type and runs its `execution` method. Each client web service contains a static `ProgrammerController` singleton object which indirectly, and with collaboration of other classes, controls the plugging and the execution of the distributed application into the object model.

![Service Class Diagram](image)

Figure 2: Client Service Class Diagram

The following Figure shows a simplified class diagram of the system components on the client side.
Each program controller has access to mechanisms which enable the client to install or uninstall programs. There is a one-to-many aggregation relationship between *programController* Class and *program* class. The same type of relationship exists between *program* class and *Group* class, *Group* Class and *Peer* class and *Group* class and *Instance* class.

![Class Diagram of Client Collaboration Objects](image)

Figure 3: Class Diagram of Client Collaboration Objects

It is possible for a program to be executed on a single client for a number of times. Since in the process of first loading, we store the binary DLL data into an object from *program* class, it is not required
that a web service client load the program again for the next execution. The only thing that is needed to be done is to create a new execution group and exchange its information between colleague peers as well as number of processes, \textit{ProgramOID} and the program \textit{rank}. Therefore, number of objects from \textit{peer} class inside \textit{group} object is equal to the number of processes. Once a program is assigned to a client web service, it informs the server web service the \textit{programOID}, \textit{groupOID} and the \textit{rank} of the program it is currently executing. \textit{Peer} class has some mechanism to communicate with server web service and request the URLs of the clients, executing a certain program in a certain group and with a certain rank. This feature provides fault tolerance capability for the system. It is very common in a P2P network that, due to network failure or program crashing, a peer leaves the network unexpectedly. It is quite possible in our system to replace a peer with another peer or to build up backups for a certain task.

On the other hand, a program may have two or more running instances of a distributed program in case the number of processors is fewer than the number of processes required. This justifies the one-to-many relationships between \textit{Group} classes and \textit{Instance} classes.

Once all the information required for the execution of a program is exchanged between colleague peers, they can create an object from the class in which it is inherited from \textit{DistributedApplication} class.
through using Reflection method. They can also plug it into the running object of the related Instance object. The peers then run the execute method of the running object.

The two other OutPipe and InPipe classes can be used to send and to receive information and implement Barrier functions.
Chapter 3
System Features and Their Implementation

This chapter addresses the basic features of the system. Programmers can write their own parallel and distributed programs in two ways. They can use main MPI like interfaces that were built into the system or benefit from a Remote function calling mechanism that is added to the system as well. Each of these features will be explained in detail as we move forward in the following sections.

3.1 Implementation of MPI Protocol

We implemented some of the main functions in MPI. These functions were implemented as part of the DistributedApplication class. As a result, once a class inherits from this class, it can easily get access to these MPI functions. Send and Receive methods are the main functions in MPI. Other functions can be easily implemented by using Send and Receive methods. The functions implemented are as follows:

- **Send**: Only an Asynchronous version of this function can be implemented. To satisfy fault tolerance feature, the Send method should be implemented asynchronously. Once a send method is called, only the value to be sent is kept in the local client. This
value can be accessed by calling *Receive* Method in another client and through the web service that hosts the distributed application object. Once a client leaves the overlay network, another peer is replaced and the peer restarts the program. Since there is no need for any synchronization between the sender and receiver peers, it only affects one peer.

- **Receiver**: Both synchronous and asynchronous versions of *Receive* were implemented. Once the *Receive* Method is called, the client gets the URLs of peers working on that application in a certain group and with a certain rank from server web service. It sequentially connects to the corresponding peers, and once the result from a corresponding *Send* is ready, it returns the value and skips the rest. If the value is not ready in any of the corresponding peers, it waits for a certain amount of time and tries it again. If all the corresponding peers fail to respond because of the network failure or crash, it requests a new peer as a replacement.

- **Broadcast**: Broadcast sends a message to all peers from a root peer on the one hand, and, on the other hand, other peers can receive from the root peer. Another version of Broadcast was also implemented that gets a list of peers’ rank and only broadcasts the message to those peers.
- **Reduce:** It links all the peers to a root peer and, in return, the root peer receives information from all other peers. Then, an operation is applied to the results received from each one of the peers and, thus, returns the total result value. In standard MPI Reduce function, it is possible to apply only a few operations. In our implementation, it is possible for a programmer to define the operation through defining a delegate. Delegate is a mechanism in .Net that allows having a pointer to a function. We, furthermore, implemented another version of Reduce that only applies to a list of peers similar to the one we implemented for the Broadcast. The Code Below shows the Signature of Reduce method:

```csharp
public delegate object OperatorDelegate(object[] objs);
public object Reduce(int ToRank, object obj, OperatorDelegate opDLG)
```

- **Barrier:** A barrier is a method for synchronization of processes. In other words, it is a means to stop peers from racing. Barrier can easily be implemented by our *Send* and *Receive* functions. The peers only need to send and to receive from one another. For instance, consider a group of 5 peers. If peer 1 sends a message to peer 2 and then receives from peer 5 and peer 2 sends to peer 3 and receives from peer 1 and ... and peer 5 sends to peer 1 and receives from peer 4, then all these peers are said to have
been synchronized. To prevent interferences of barriers with regular sends and receives, a separate send and receive pipe object is built into the system. In our implementation, we provided the possibility to have barriers for a certain group of peers similar to the approach we used for Broadcast and Reduce.

- **Terminate**: Terminate method synchronizes all the peers and then releases the resources they have used. It is similar to the Finalize function in Standard MPI.

### 3.2 Implementation of Remote Function Calls

Beside the MPI functions, another method of parallel and distributed execution was implemented in our system. These methods are fundamentally based on MPI send and receive functions. Nevertheless, it equips the programmers with the ability to call a method on a peer with a certain rank. The Methods below are implemented:

- **InvokeMethod**: This method calls a function in a certain peer and returns the result. This function must be succeeded with RunMethod in the receiver side in order to be operated. Two other functions, namely BeginInvokeMethod and EndInvokeMethod were implemented to provide parallel
execution of the method. The code below shows the implementation of this method:

```csharp
public object InvokeMethod(int ToRank, string MethodName, object[] Parameters)
{
    FunctionSignatureClass FuncSig = new FunctionSignatureClass();
    FuncSig.MethodName = MethodName;
    FuncSig.Parameters = Parameters;
    Send(ToRank, FuncSig);
    return Recieve(ToRank);
}
```

- **RunMethod**: It executes the method received from another peer. This method uses Reflection to execute a given function name received from a peer. **BeginRunMethod** and **EndRunMethod** provide Asynchronous calling of this function.

The Code below shows how this method is to be implemented:

```csharp
public void RunMethod(int fromRank)
{
    FunctionSignatureClass FuncSig = (FunctionSignatureClass)Recieve(fromRank);
    Type mytype = this.GetType();
    MethodInfo Mymethod = mytype.GetMethod(FuncSig.MethodName);
    object result = Mymethod.Invoke(this, FuncSig.Parameters);
    Send(fromRank, result)
}
```
3.3 Sample Programs

In this section, we show how a sample program can be written. For the \(|q|<1\), the following formula is convergent. We aim to calculate the following expression for a given input.

\[
\frac{1}{1-q} = 1 + q + q^2 + q^3 + \ldots
\]

The following code is written using MPI functions. As it is observed, the code MyApp inherits from DistributedApplicationClass. We Override the Execute Method. In this implementation we used MPI send and receive functions.

```csharp
public class MyApp:DistributedApplicationClass
{
    public override void Execute()
    {
        if (Rank == 0)
        {
            double input = 0.5;
            for (int i = 1; i < NoOfProcesses; i++)
            {
                Send(i, input);
            }
            double output = 1;
            for (int i = 1; i < 5; i++)
            {
                output += Math.Pow(input, Rank * i);
            }
            for (int i = 1; i < NoOfProcesses; i++)
            {
                output += (double)Recieve(i);
            }
            MessageBox.Show(output.ToString());
        }
    }
}
```
else
{
    double input = (double)Recieve(0);
    double output=0;
    for (int i = 1; i < 5; i++)
    {
        output+= Math.Pow(input, Rank*i);
    }
    Send(0, output);
}
Terminate();

The following code shows the same calculation with Remote Method Calling approach. AsyncRes value stored here is used to keep the state reference of Asynchronous method invocation and is later used for EndInvoke method.

public class MyApp:DistributedApplicationClass
{
    public override void Execute()
    {
        if (Rank == 0)
        {
            double input=0.5;
            IAsyncResult[] AsyncRes=new IAsyncResult[NoOfProcesses];
            for (int i = 1; i < NoOfProcesses; i++)
            {
                AsyncRes[i]=BeginInvokeMethod(i, "f",
                                                new object[ ] { input, i });
            }
            double output=1;
            for (int i = 1; i < 5; i++)
            {
                output+= Math.Pow(input,Rank*i);
            }
            for(int i = 1; i<NoOfProcesses; i++)
{ 
    output += (double)EndInvokeMethod(AsyncRes[i-1]);
} 
MessageBox.Show(output.ToString());
}
else
{
    RunMethod(0);
}
Terminate();
}
public double f(double x,double y)
{
    double output = 1;
    for (int i = 1; i < 5; i++)
    {
        output+=Math.Pow(x,y*i);
    }
    return output;
}
Chapter 4  
Advanced Features

The initial solution is based on a centralized network. All the peers are informed of a central server concerning their execution objects. This system works in an environment with limited number of peers. However, system scalability is highly desired. To achieve this goal, a pure decentralized algorithm is necessary. For the present, DHT Algorithms are available and can provide us with the mechanism of searching resources in a satisfactory manner. Algorithms like CHORD [5], Pastry [3], Tapastry [2] and CANS [6] are available. They enable us to seek and to search for resources over the overlay network by $O(\log(n))$ hops. In our studies, we resorted to CHORD Algorithm to create a highly scalable peer to peer network. One of our main queries is to know which peers work on an application with a given groupOID and rank. It is possible to distribute these resources among multiple servers by using SHA1 hashing function used in CHORD algorithm. Since our client services are decoupled from our server side services, it is possible to base a scalable network by changing the server side codes only.

The second improvement is to add the check-pointing feature such that we can have a better fault tolerance system. In the initial
implementation, once a peer fails, another peer is replaced and the new peer immediately restarts the process. It is possible in .Net to serialize certain types of objects and to store and to distribute them in an XML or a binary format. However, storing of the execution state of an object needs more programming efforts. To develop a more adequate technique, we implemented a CheckPoint function that stores the execution states and loads the latest state in case it is necessary.

4.1 Distributed Hash Table (DHT) Algorithms

DHT refers to the group of decentralized distributed systems that distribute and balance information between collaborative nodes. A key is assigned to the information and a certain node becomes responsible for managing and storing of data about that information. The role of these DHT algorithms is to support a scalable, decentralized and reliable system that performs searches for the nodes that contain these data. It, therefore, makes the access to the information easier in an acceptable amount of time. In our implementation, we benefited from CHORD algorithm that provides possibilities for finding data and resources in the network in $O(\log (n))$. In the following section, we review the details concerning this algorithm and how it is supposed to be implemented.
4.2 Chord Algorithm

Using Chord lookup algorithm nodes are placed in a circle. The circle can not have more than \( 2^m \) nodes which in the case of using a hash algorithm like SHA1 \( m \) it will be 160. In this ring each node has an ID from 0 to \( 2^m - 1 \). This ID is assigned to the node by finding the SHA1 hash value of its address. For example:

\[
\text{Node identifier} = \text{SHA-1(IP address)}
\]

\[
\text{IP}=25.1.10.1 \quad \text{SHA-1} \quad \text{ID}=1425754228352
\]

In this ring each nodes knows \( m \) nodes. Each node stores and updates the list of these nodes in a list data structure called "Finger Table". In this finger table, nodes store the list of successor of \( n + 2^i \) keys where \( 0 \leq i < m \) and \( n \) is the ID of the node. Moreover, each node knows the successor and predecessor nodes of itself. The successor to a node or a key is the next node in the identifier circle in the clockwise direction. The predecessor of a node or a key is the next node in the ID circle in the counter-clockwise direction. Figure 4 shows how each node knows \( m \) other nodes.
When an info is searched in the network, the info is converted to a key which is $Key=SHA1(Info)$. Each Node has the responsibility of maintaining data about the Key between its predecessor and its ID. Therefore, the function $GetSuccessor(Key)$ is defined and it can find the successor of a key with $O(log(n))$ hops in the network. Figure 5 shows a simplified Chord network with maximum of 32 nodes. Each node has a finger table with the $\log_2 32 = 5$ length that stores list of successors of $n + 2^i$ keys where in this case $0 \leq i < 5$. 

![Figure 4: Nodes Placement in CHORD Algorithm](image)
Figure 5: Nodes and their Finger Table in CHORD

For example, for the node number 1, it keeps the address of the successors of keys 2, 3, 5, 9 and 17 which point to the nodes with IDs of 4, 4, 7, 12 and 20. Node 1 is succeeded by node 4. Once info is searched in the network, it is first converted into the key. Then the key is searched in the network. Suppose we want to find the successor for the Key=14 from node number 1. In this case, node 1 first looks for the key which might be located between itself and its successor. Once it matches the condition, it returns to its successor as result. In this case, it does not match these criteria. Thus, it looks for the best node that
might know the answer to the question that the successor of 14 is. In the node, 1 finger table node 12 exists that is less than 14 so the question is forwarded to node 12. Node 12 checks if the key is between itself and its successor which is, in this case, it is node 15. Since it matches the condition, it returns to the address of node 15 as the successor of 14 to node 1.

Once a node joins the network it first looks for its successor and predecessor. Then it informs its predecessor and successor that it has joined the network and asks the successor to exchange the keys between its ID and its predecessor ID.

Following that, it completes its finger table by querying its successor node. Since joining and leaving of the network may cause inconsistency in other nodes finger tables, each node should refresh its finger table periodically. Moreover, unexpectedly node failure is handled through redundancy in the network and backup pointers to reestablish damaged links. This is done through storing n’s immediate connected successor and predecessor nodes and through backing up their execution information. In the next section, we will study main steps of the algorithms and their implementations.
4.2.1 Joining and Leaving of Nodes To/From Network

It is necessary for a node to know another node as a gateway in order to establish connection and to join the chord network. The other required steps are to find the successor and predecessor nodes and notify them. Following that, the node refreshes its finger table, its successor and predecessor list through another thread of control. This is done periodically to ensure stability and robustness of the system. At the end, the node exchanges the indexed data with its successor. The code below shows steps necessary for stabilizing a node.

```csharp
public void JoinNetwork()
{
    if (!isJoinedNetwork)
        return;

    RemoteNode.Service myService = new RemoteNode.Service();
    myService.Url = gatewayURL;
    string tempUrl = myService.GetSuccessor(MyNodeNo.getBytes());
    successorURL = tempUrl;
    myService.Url = successorURL;
    tempUrl = myService.GetPredecessorURL();
    myService.UpdatePredecessor(myURL);
    predecessorURL = tempUrl;
    myService.UpdateSuccessor(myURL);
    isJoinedNetwork = true;
    RefreshLookupTableAsync(true);
}```
RefreshSuccessorListAsync(true);
RefreshPredecessorListAsync(true);

//exchangedata
ExchangeInfoJoin(myService);

Figure 6: Sequence Diagram for Joining of a Node to the Network

When a node leaves the network it notifies its successor and predecessor and exchanges its information to its successor.

public void LeaveNetwork()
{
    if (!isJoinedNetwork)
        return;
    isJoinedNetwork = false;
RemoteNode.Service myService = new RemoteNode.Service();
myService.Url = successorURL;
myService.UpdatePredecessor(predecessorURL);

myService.Url = predecessorURL;
myService.UpdateSuccessor(successorURL);

//ExchangeInfo
ExchangeInfoLeave(myService);

successorURL = myURL;
predecessorURL = myURL;

Figure 7: Sequence Diagram for Leaving of a Node from the Network
4.2.2 Get Successor Function

This is one of the most important functions in the Chord Algorithm. This function is not only used to find a successor of a node but it is also used to find information and to create finger tables. It plays a vital role and is very operative and critical in stabilizing the network. This function first checks whether the key number is between the node ID and its successor ID. If it is between these two IDs, then it returns successor of the node as result. Otherwise, through using the node’s finger table, it tries to ask the result from the best node that is closer to the key and at the same time less than the key. This is possible by forwarding the key to \textit{getsuccessor} function of that best node. This function recursively and with the most $O(\log (n))$ hops is able to find the result.

public string GetSuccessor(byte[] valueBytes)
{
    BigInteger value=new BigInteger(valueBytes);
    if ((gatewayURL == myURL) && (successorURL==myURL) && (predecessorURL==myURL))
        return myURL;
    else
    {
        if ((MyNodeNo < SuccessorNodeNo) && (PredecessorNodeNo<MyNodeNo))
            {
                if ((value >= MyNodeNo) && (value < SuccessorNodeNo))
                {
                    return successorURL;
                }
            }
else if ((value >= PredecessorNodeNo) && (value < MyNodeNo))
{
    return myURL;
}
else
{
    //forward it.
}
}
else if ((MyNodeNo < SuccessorNodeNo) &&
    (PredecessorNodeNo > MyNodeNo))
{
    if ((value >= MyNodeNo) && (value < SuccessorNodeNo))
    {
        return successorURL;
    }
    else if ((value >= PredecessorNodeNo) || (value <
        MyNodeNo))
    {
        return myURL;
    }
    else
    {
        //forward it.
    }
}
else if ((MyNodeNo > SuccessorNodeNo) &&
    (PredecessorNodeNo < MyNodeNo))
{
    if ((value >= MyNodeNo) || (value < SuccessorNodeNo))
    {
        return successorURL;
    }
    else if ((value >= PredecessorNodeNo) && (value <
        MyNodeNo))
    {
        return myURL;
    }
    else
    {
        //forward it.
    }
}
private static string ForwardToBestNode(byte[] valueBytes, BigInteger value)
{
    string ForwardUrl = SuccessorURL;
    int MinIndex = -1;
    int MaxIndex = -1;
    BigInteger Minn = new BigInteger(-1);
    BigInteger Maxx = new BigInteger(-1);
    for (int i = 0; i < FingerTable.Length; i++)
    {
        if ((FingerTable[i].Successor < value) && (FingerTable[i].Successor > Minn))
        {
            Minn = FingerTable[i].Successor;
            MinIndex = i;
        }
        if (FingerTable[i].Successor > Maxx)
        {
            Maxx = FingerTable[i].Successor;
            MaxIndex = i;
        }
    }
    if (MinIndex >= 0)
    {
        ForwardUrl = FingerTable[MinIndex].Url;
    }
    else if (MaxIndex >= 0)
    {
        ForwardUrl = FingerTable[MaxIndex].Url;
    }
    else
    {
        ForwardUrl = successorURL;
    }
    return Forward(valueBytes, ForwardUrl);
}
private static string Forward(byte[] valueBytes, string ForwardUrl)
{
    RemoteNode.Service myService = new RemoteNode.Service();
    myService.Url = ForwardUrl;
    return myService.GetSuccessor(valueBytes);
}

4.2.3 Refreshing of Finger Table

Function RefreshLookupTable periodically refreshes the finger table by querying the successor node for the successor nodes of the given keys which, as it was mentioned earlier, are $n + 2^i$ keys where in this case $0 \leq i < 160$ and $n$ is the node ID.

public void RefreshLookupTable(bool Infiniteloop)
{
    while (isJoinedNetwork && Infiniteloop)
    {
        BigInteger two = new BigInteger(2);
        BigInteger TwoPower160 = new BigInteger();
        TwoPower160 = two.Power(160);

        for (int i = 0; i < FingerTable.Length; i++)
        {
            BigInteger tempval = new BigInteger();
            tempval = MyNodeNo + two.Power(i);
            if (tempval >= TwoPower160)
            {
                tempval = tempval - TwoPower160;
                FingerItem item = Successor(tempval);
                FingerTable[i] = item;
            }
        }
        Thread.Sleep(120000); // Refresh it periodically
    }
}
4.3 Check-Pointing Feature

It is so common in a peer to peer environment that nodes leave the network and, at the same time, many of these nodes may be in the middle of processing of a program. In these cases, another node should replace the left node and start the processing of a program. In cases where the program execution is small it is feasible to restart the processing of the program from scratch, however, in cases where a program execution takes lots of CPU time it is essential to store the state of a program in the network so that the replaced node continues the program from a state saved.

In .Net framework, it is possible to serialize an object to a binary or XML format. To achieve this, one of the methods is to use [Serializable] attribute tag. It is necessary to tag also all referenced fields in the class which also need to be serializable. Program below shows an example of a serializable class:

```
[Serializable]
public class Class1
{
   protected class2 TheObject;
   ...
}
```
[Serializable]
public class Class2
{
    protected Int ID;
    protected String Name;
    . . .
}

System.Runtime.Serialization.Formatter namespace in .Net provides us with XML and Binary formatter that are able to Serialize and Deserialize the classes with the serializable tag. It is important to notice that, here again, Reflection is used internally to provide us with this feature. Codes below shows the use of BinaryFormatter to Serialize and Deserialize of a class:

public byte[] ToBinary()
{
    BinaryFormatter bf = new BinaryFormatter();
    MemoryStream ms = new MemoryStream();
    bf.Serialize(ms, this);
    ms.Seek(0,0);
    return (ms.ToArray());
}

public Object ToObject(byte[] binaryData)
{
    BinaryFormatter bf = new BinaryFormatter();
    MyBinderClass Binder = new MyBinderClass();
    Binder.BindingType=this.GetType();
    bf.Binder=Binder;
    MemoryStream ms = new MemoryStream(binaryData);
    return bf.Deserialize(ms);
}
Although it is possible to easily save the object state of a program nevertheless saving the execution state of an object needs more programming effort. To achieve this goal, two status objects have been built into the *DistributedApplication* class that, along with the Checkpoint function, provide us with Check-Pointing functionality.

```csharp
public bool CheckPoint()
{
    bool result = (RunningStatus == Status2);

    if (result == true)
    {
        byte[] ObjectBinary = ToBinary();

        // store it in the right server
        ProgramControllerClass ProgramController =
            this.Instance.Group.Program.ProgramController;
        ProgramController.StoreStatus(ProgramOID, GroupOID,
            Rank, RunningStatus, ObjectBinary);

        RunningStatus++;
        Status2++;  
    }
    else
    {
        Status2++;  
    }
    return result;
}
```

The function saves the status of a program and specifies if the code block can be executed. By using *IF* statement blocks of a program is specified. Example:
public override void Execute()
{
    if (Rank == 0)
    {
        if (CheckPoint()) //Code Block 1
        {
            Send(1, 10);
        }
        if (CheckPoint()) //Code Block 2
        {
            Send(2, 15);
        }
        ...
        if (CheckPoint()) //Code Block n
        {
            FinalResult=Receive(3)
        }
    }
    ...
}
Chapter 5
Conclusion

Peer to Peer networking provides us a faster and better way of program execution through sharing power of processors and network resources. The goal of our system is to provide resources which are equally accessible to the counterparts. Utilizing MPI, the programmers find it possible to write a fully-distributed and parallel application. Our system provides hosting of them. It is even possible with Web services to set up our environment on resources available on the internet with lower costs.
References


[13] Seti@Home Home Page.

[14] MPI Forum
    http://www.mpi-forum.org/docs/docs.html

[15] Microsoft Web Services and Other Distributed Technologies
    Developer Center
    http://msdn.microsoft.com/webservices/

[16] Web Services website
    http://www.w3.org/2002/ws/

[17] Folding@home website
    http://folding.stanford.edu/

[18] Einstein@Home web site
    http://www.einsteinathome.org

[19] Extrem Web website
    http://www.xtremweb.org

Appendices

Appendix A: Server and Client Services Class Diagrams (Final Design)

![Server Service Class Diagram]

Figure 8: Server Service Class
Figure 9: Client Service Methods and Attributes
Appendix B: Business Classes Diagrams

Figure 10: Business Classes Class Diagram

Figure 11: ProgramController Class Methods and Attributes
Figure 12: DistributedApplication Class Methods and Attributes
Figure 13: Program Class Methods and Attributes

Figure 14: Group Class Methods and Attributes
Figure 15: Peer Class Attributes

Figure 16: Instance Class Methods and Attributes

Figure 17: OutPipe Class Methods And Attributes
Figure 18: InPipe Class Methods And attributes
Appendix C: Graphical User Interfaces

Figure 19: Management Page for Monitoring Server Service

Figure 20: Displaying of List of Items in Finger Table
Figure 21: User Interface for Managing Client Service, Loading and Executing of Programs