Building a Peer to Peer Message Passing Environment by Utilizing Reflection in .NET

Behzad Parviz  Kamyar Miremadi
Department of Computer Science
California State University, Los Angeles
Email: {bparviz, mkamyar}@calstatela.edu

Abstract

This paper describes a peer to peer solution for message passing interface (MPI) protocol. MPI is a standard protocol that is widely utilized in many of the 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 it available for programmers to write a quick and easy to use parallel and distributed program as well as internal classes that is used for our communication and execution model.

Keywords: Peer to Peer, Parallel and Distributed Programming, Message Passing Interface, Web Service, .Net, Reflection.

1. Introduction

Nowadays, many project based on peer to peer computing (one form of grid computing) are conducted around the world. Grid computing (or the use of a computational grid) is applying the resources of many computers in a network 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 to those which need access to large amounts of data. Projects like SETI@HOME [13] use the similar approach to do large number of computation by using volunteer computers the internet.

However, it needs to be mentioned that many of these approaches are limited to a single functionality and do not provide multi purpose environments for any type of applications. MPI [14] is a communicating programming model for general purpose 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. For achieving this purpose, the programmer needs to generalize and override a class inside our class libraries. This class furnishes us with an 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. Depending on these two options, the program is compiled dynamically and will be dispatched to the required number of nodes for parallel execution. 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 to programming language paradigm as 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, whereas, Linguistic Reflection refers to the ability of program to alter its data structure and
to add new coding. In modern programming language and environment like Java and .NET it is possible to use Linguistic Reflection. Linguistic reflection gives a programming system the ability to generate new program fragments and incorporate them into the ongoing computation.

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 programming efforts. However, .Net Framework is rich of many client-server communication technologies such as .Net Remote Services and Web Services such that enables us to get access to the intended goal easily.

2.1 Utilizing of .Net Web Services

In our model we utilized web services. Web services have some advantages over the Remote services and seem to be more efficient:

- Web services are language and platform independent. Since Web services use WSDL, which is an XML format for describing the service and SOAP as an exchanging message protocol, Of course, it is possible to define and consume web services in different languages.
- By Utilizing HTTP protocol, web services can work through the firewalls.
- Web Services is loosely coupled so they provides 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 applications and can easily be utilized in our system. It reduced 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 (RPC) through transferring requests and, in return, through receiving responses. In another word, it seems not to be possible to manage an object in a server from a client without providing some persistent methods. In our implementation, we used singleton patterns [9]. In some cases, we used a personal database (SQL Server Express) to store these objects. 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 run-time. 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 provides 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 from C# and VB.NET language.

The code below shows how we benefit from Reflection to load an assembly and create and reference an object at runtime.

```csharp
public DistributedApplicationClass CreateP2PObject(byte[] AssemblyCode)
{
    Assembly Assm = Assembly.Load(AssemblyCode);
    object obj = null;
    foreach (Type typ in Assm.GetTypes())
    {
        obj = Activator.CreateInstance(typ);
        if (obj is DistributedApplicationClass)
        {
            break;
        }
        return (DistributedApplicationClass)obj;
    }
}
```

In our programming model, each application must inherit from DistributedApplication Class. We will discuss about 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 DistributedApplication Class, it can create an object and return it as the result.

It is also possible to call a method at runtime.
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 MethodInfo Class looks for the method with the given name, invokes the method on current object and sends the result back to the RunMethod 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 which application, in which group and with what rank). The following figure shows the Methods that implemented in the server web service.

![Figure 1: Server Service Class Diagram](image)

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 execute method. Each client web service contains a static ProgrammerController singleton object which indirectly, and with collaboration of other classes, controls the plugging and execution of distributed application into the object model.

![Figure 3: Class diagram of client collaboration objects](image)

Figure below shows a simplified class diagram of the system components on the client side.

Each program controller has access to mechanisms which enables the client to install or uninstall programs. There is a one-to-many aggregation relationship between programmerController 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.

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
needed for a client web service to 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, programOID and the program rank. Therefore, number of objects from peer class inside 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 programOID, groupOID and the rank of program it currently executes. 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 the certain rank. This feature provides fault tolerance capability for the system. It is very common in a P2P network that a peer leaves the network unexpectedly due to network failure or program crashing. It is possible in our system to replace a corresponding peer 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 network failure or crash, it requests a new peer for replacement.

- **Receive**: 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 network failure or crash, it requests a new peer for replacement.

- **Broadcast**: Broadcast sends a message to all peers from a root peer and on the other hand, other peers receive from the root peer. Another version of Broadcast was also implemented that get a list of peers' rank and broadcast the message to only those peers.

- **Reduce**: It sends from all the peers to a root peer and, in return, the root peer receives from all the peers. Then an operation is applied to the results received from each one of the peers and 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 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:

```java
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 implemented by our **Send** and **Receive** functions. The peers only need to send and receive from each other. 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 peers 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 objects 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.

### 4. 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**: 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 implementation of this method:

```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);
}
```

### 5. Sample program

In this section, we show how a sample program can be written. For the $|q| < 1$ the following formula is convergent. We want 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 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.

```csharp
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;
}

7. Future Work

Currently our solution is based on a centralized network. All the peers inform a central server of 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 for having better fault tolerance system. For the time being, once a peer fails, another peer is replaced and restarts the process. It is possible in .Net to serialize certain types of objects and store and distribute them in a XML or binary format. However storing of the execution state of an object needs more programming efforts.

8. Conclusion

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

References


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

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