CHAPTER 4

.NET REMOTING AND CONTINUOUS SERVICE MODEL
FOR ON-LINE ECONOMIC LOAD DISPATCH

4.1 INTRODUCTION

As power systems become large and more complex, effective on-line power system algorithms in a distributed environment are required in order to reduce the cost of power system operation and control. On the other hand, the advance of information technology such as distributed object technologies is progressing rapidly. Economic load dispatch is one of the most important on-line power system problems that to be solved in a distributed environment where the power system substations are located geographically wider. A remoting architectural model and a mobile agent model for on-line economic load dispatch in a distributed environment are explored in this chapter.

4.2 REMOTING ARCHITECTURAL MODEL FOR MULTI-AREA POWER SYSTEM ON-LINE ECONOMIC LOAD DISPATCH

The main objective of the proposed model is to construct a distributed environment through which the economic load dispatch solutions of multi-area power systems can be monitored and controlled. A single-server/multi-client architecture, which enables the neighboring power systems to
access the remote Economic Load Dispatch (ELD) server at any time with their respective data and to get the economic load dispatch solutions from the remote server, has been proposed. A .NET Remoting architectural model has been implemented in such a way that for every specific period of time, the remote server obtains the system data simultaneously from the neighboring power systems which are the clients registered with the remote ELD server and the optimized economic load dispatch solutions with power loss from the server have been sent back to the respective clients. The economic load dispatch server creates a new thread of control for every client’s request and hence complete distributed environment is exploited.

This proposed work outlines a new approach to develop a solution for economic load dispatch analysis by the way of distributed computing. A .NET remoting client-server architecture overcomes the difficulties associated with sequential computation and it can be easily deployed. Remoting provides a powerful and easy way to the power system clients to communicate with remote ELD server at different domains. The .NET framework provides a number of services, including activation and lifetime support, as well as communication channels responsible for transporting messages to and from remote applications. Remoting uses built in security by providing a number of hooks, which allows channel sinks to gain access to the power system data in the form of serialized stream before it is transported over the channel and hence the distributed economic load dispatch through this proposed model secures the safety of the server as well as power system data. In this present work, a distributed environment has been set up using .NET Remoting to estimate and to monitor economic load dispatch solutions for different sub-systems of an integrated power system. Each sub system has been considered as a power system client
and hence multi power system clients – single ELD server model is implemented. In this model the tie line power flow for each area is assumed to be constant and treated as load for each sub system. These power system clients are interconnected with an ELD server as shown in Figure 4.1. When there is a call for a method located in a remote object, Figure 4.1 shows the flow of .NET Remoting model.

Figure 4.1 .NET REMOTING Architecture for Economic load dispatch
A client computer basically does the distributed power system monitoring through a GUI for every specific period of time and frequently exchanges data with the server. The server does the economic load dispatch computation and then distributes the results. Chronologically the server process should be started first, so that it can take the initiative to set up a connection link. It then starts waiting till it receives a connection request from the client. A client can register itself with the remote object (server object), just by invoking the registration procedure on the server object, when it needs a service from it. The remote object obtains the necessary data from the registered client objects and responds back to them respectively with the results. This total process can be automated by making the server get the input data for every specific period of time.

4.2.1 .NET REMOTING Data flow Model

In the proposed model, each neighboring power system is considered as a client remote object and registers as a new instance with the ELD server. The proposed remoting system creates a proxy object that represents the server class and returns a reference of it to the client object. When a client calls a method on an object that represents the remote object, the remoting infrastructure fields the call and it checks the type information and sends the call over the channel to the server process. Channel is a type that packages the power system data into a block of bytes that can be communicated through the HTTP protocol. The HTTP channel transports messages to and from remote objects using the Simple Object Access Protocol (SOAP). All messages are passed through the SOAP formatter, where the message is changed into XML and serialized, and the required SOAP headers are added to the stream.
This process of marshalling presents the entire economic load dispatch data in a suitable format for transporting from one system to another. When the channel sends the information to the economic load dispatch server, a listening channel in the server side picks up the request and unmarshals the parameters. This receiver locates the object to be called and then calls the desired method with those parameters. When the method returns, server remoting system captures, marshals the return value and sends the marshalled economic load dispatch results as packets on to a marshal stream and thus sends the results to the client channel. The client remoting system returns the results of the call to the client object. This data flow model is shown in Figure 4.2.

Figure 4.2 Remoting data flow model
4.2.2 Versioning service model of ELD server

Maintaining large applications like economic load dispatch across different versions is a complex problem. This is especially true when the ELD server component undergoes significant changes between versions. .NET Remoting solves these problems by versioning assemblies, thereby allowing applications to run only with the versions they were built and tested. In this proposed method, the economic load dispatch server is made to execute different versions of economic dispatch service in Common Language Runtime (CLR) environment. The steps involved in the side-by-side versioning of economic load dispatch server are as follows:

ELD server component has given a version number as part of its identity. The version number is stored in the assembly manifest along with other identity information like relationships and identities of other assemblies connected with the application. In the absence of a policy file in the assembly, the economic dispatch server should continue to work with the CLR version it was built with. If a policy file is present specifying that two different versions of an assembly like System.Runtime. Remoting are compatible, fusion will automatically load the latest compatible version. It is therefore important to ensure that only strong assembly names are used in configuration files when referring to CLR assemblies.

Economic load dispatch application is made to run side-by-side by using strong names when calling Assembly.Load or when specifying power system client’s transport sink providers. Any changes or up-gradation of the
economic load dispatch server logic results in the side-by-side versioning service and upgraded version is made available for clients.

4.2.3 .NET REMOTING based economic load dispatch monitoring algorithm

Each client object registers with the ELD server component. The server uses the remote client reference to invoke its method to obtain the economic load dispatch data and then provides the service through its methods. The server object uses a single thread of control to distribute the economic load dispatch solutions simultaneously to the clients registered with it. The proposed model is dynamic which allows a new power system client to register with the ELD server object at run-time and get serviced. Economic load dispatch server and clients have to store in them the necessary object codes required for economic load dispatch calculations. Subsequently, the following steps are to be carried out.

- Start economic load dispatch server.
- Economic load dispatch server creates its own proxy object and run the proxy object at specific port number.
- Start power system client by loading the server’s side DLL in the assembly.
- Client registers with the server by invoking the appropriate method at the remote object.
• Server uses the client’s reference to receive the power system data from the client.

• Server computes the economic load dispatch result and returns it to the client.

• Client obtains the result from the server through server side HTTP channel and provides a view of the result through a specified object.

• For every specific period of time, server automatically receives system data from the client, thereby providing an automatic economic load dispatch monitoring.

4.2.4 Implementation

Many of today’s power system applications employ distributed object technology for scalability and reliability. Remoting is a new technology supporting this goal, allowing objects to communicate across application domains with minimal overhead. The remoting architecture abstracts as much of the underlying communications plumbing associated with distributed computing. It’s easy to setup communication between remote power system distributed objects and communicate as if they were in the same process space.

Remoting can be approached at many different levels. One of the significant differences between remoting and other distributed object technologies is its architectural extensibility. The remoting architecture is very flexible, enabling the capability to extend power system application by adding custom components that participate in the communication process.
An integral feature of remoting is lifetime management of remote economic load dispatch server components through leases. A power system client communicates with a lease manager to control the life time of these components.

At its most basic level, remoting is the capability to communicate with components in separate AppDomains. Figure 4.3 shows a simplified view of two objects communicating via remoting – a power system client component in AppDomain A communicates with an economic load dispatch server component in AppDomain B. The application domain is an execution environment within the remoting architecture. It provides major benefits of reliability and security of power system data communication.

![Figure 4.3 Basic remoting architecture for Economic load dispatch](image-url)
Since remoting supports multi-tiered as well as distributed architectures, the server component in AppDomain B could easily be extended to any power system client to service the economic load dispatch. Furthermore, a remote component could be located in another process on either the same or a different machine. Once an ELD component is set up, the underlying plumbing required to maintain remote communication is hidden by the remoting architecture.

4.2.5 Remoting economic load dispatch server

A remoting ELD server object is an instance of a class that inherits from MarshalByRefObject. Figure 4.4 shows the implementation of the remote ELD server object. The EldServer class in Figure 4.4 inherits MarshalByRefObject, which supports the basic functionality of a callable ELD component over the proposed remoting architecture. This class is implemented as a DLL, negating the need for a Main() method in the economic load dispatch server. The only method in this class is the calculateEconomicLoadDispatch() which receives the economic load dispatch data as a string argument and calculates the economic load dispatch for the given data. The calculateEconomicLoadDispatch() method sends the economic load dispatch results to the power system client as a single string.
using System;
using System.IO;
namespace EldServer
{
    public class RemoteELDImpl : MarshallRefObject, RemoteELDInterface
    {
        public String calculateEconomicLoadDispatch(String pdata)
        {
            //Economic load dispatch calculation
        }
    }
}

Figure 4.4 Implementation of ELD server object

The remote ELD component requires a configuration file named web.config to specify necessary operating parameters. The web.config file for the remote ELD server component is shown in Figure 4.5.

```xml
<configuration>
    <system.runtime.remoting>
        <application>
            <service>
                <Wellknown mode="SingleCall"
                type="EldServer.RemoteELDImpl, EldServer,
                    Version = 1.0.941.28888,
                    Culture = neutral,
                    objectUri="ELDObject" />
            </service>
        </application>
    </system.runtime.remoting>
</configuration>
```

Figure 4.5 web.config file for ELD server object
The web configuration file shown above is located in the same directory in which the ELD server implementation is located. It contains a special section for remoting, marked by a tag name after the remoting namespace, `<system.runtime.remoting>`. Further down the hierarchy is an `<application>` tag, containing a `<service>` tag. The `<service>` section has a `<wellknown>` tag with three attributes such as mode, type and objectUri that assist in making the remote ELD object to run properly.

The attribute “mode” has two possible values: SingleCall and Singleton. These values identify basic life time issues associated with a remote economic load dispatch server object. SingleCall components are activated and live for the duration of a single call from the remote client. When a remote power system client calls the `calculateEconomicLoadDispatch()` method of the `EldServer` class, a new server object is instantiated when the call begins. When the call ends, the server object is destroyed. Furthermore, each client receives a reference to a unique server object whenever it makes a call.

The type attribute of the `<wellknown>` tag has a quoted pair of values that identify the remote ELD server object. The first value is the fully qualified name of the server class. The second value is the executable filename of the assembly to which the server class belongs to. The last attribute of the `<wellknown>` tag is the objectUri, which holds the Universal Resource Identifier (URI) of the server component.

### 4.2.6 Remoting Power system client

Designing a power system client in .NET remoting environment requires finding out the type and location of the remote server object,
initializing the configuration file and creating a reference to the remote object with the type and location information obtained earlier. The power system client that will make a call to the economic dispatch server component is implemented as shown in Figure 4.6.

```csharp
using System.Runtime.Remoting;
using EldServer;

public class RemoteClient
{
    public static void Main(String [] args)
    {
        ChannelServices.RegisterChannel(new HttpChannel( ));
        RemoteELDInterface obj =
            (RemoteELDInterface)Activator.GetObject
            (typeof(RemoteELDInterface),
            "http://localhost:1234/ELDObject.soap");
        RemoteClient rc = new RemoteClient( );
        string s = rc.readFile("pdata.dat");
        string status = obj.calculateEconomicLoadDispatch(s);
        Console.WriteLine(status);
        rc.writeFile(status);
        Console.ReadLine();
    }
}
```

Figure 4.6 Implementation of power system client
In the power system client implementation the namespaces `System.Runtime.Remoting` and `EldServer` are used. The ELD server component that will be instantiated and called is in the `EldServer` namespace. A power system client configuration file is named after the executable of the client with the `.config` extension. The executable name of the program is `RemoteClient.exe` and the configuration file name must be `RemoteClient.exe.config` and it is shown in Figure 4.7

```xml
<configuration>
  <system.runtime.remoting>
    <application name="RemoteClient">
      <client>
        <wellknown type = "EldServer.RemoteELDImpl,
                      EldServer,
                      Version = 1.0.941.28888,
                      Culture = neutral,"
        url="http://127.0.0.1:7777/ELDObject.soap" />
      </client>
      <channels>
        <channel Port=7777
        />
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

Figure 4.7  Power System Client configuration file
The first difference between the power system client configuration file and the server side web configuration file is that the `<application>` tag has a name attribute defined. This is the name of the power system client class, RemoteClient. Lower in the hierarchy is the `<channels>` tag, where a channel is defined. The `<channels>` section has `<channel>` tag with a type attribute that has a string with two values. The first value specifies what type of channel will be used to communicate with the server component. The second parameter identifies the namespace associated with the channel. Once the power system client is configured, it may obtain a reference to the remote object. This is accomplished by calling the static `GetObject()` method of the `Activator` class. The two parameters to the `GetObject()` method are the `type` and `url`, respectively that were obtained earlier. Since the reference is returned as an object type, the return value is cast to `RemoteELDInterface`. After the remote server component reference is obtained, `calculateEconomicLoadDispatch()` method is invoked by the power system client to obtain economic load dispatch solutions. The above distributed algorithm has been implemented in Windows NT based HP workstations connected in an Ethernet LAN. The results are shown in a GUI as given in Figure 4.8.

The GUI shows the economic load dispatch solution for a specific 3-generator bus power system client. When each power system client is loaded, it registers with the economic load dispatch server and hence the intercommunication between the server and client remotable objects is achieved.
Based on this proposed model, different power system clients can monitor continuous, updated and optimized economic load dispatch solutions at regular time intervals.

4.3 MOBILE AGENT MODEL FOR ECONOMIC LOAD DISPATCH

The aim of the proposed work is to construct a mobile agent model in a distributed environment through which the economic load dispatch solutions
of multi-area power systems can be monitored and controlled. In the proposed model for ELD, a mobile agent is created which enables the mobility of the economic dispatch executable code to the neighboring power system client host. A single server (main container) / multi client (sub container) based architecture has been developed which enables the power system client to access the economic load dispatch mobile agent at it’s own framework with their respective data and to get the economic load dispatch solutions. A distributed agent model has been implemented in such a way that for every specific period of time, the ELD agent migrates from one power system client to another to obtain the system data and the optimized economic load dispatch solutions with power loss have been sent back to the clients in a heterogeneous environment. The ELD agent creates a new thread of control for every client’s request and hence complete distributed environment is exploited. This model outlines a new methodology for solving economic load dispatch analysis in a distributed environment.

The mobile agent framework provides a number of advantages including the saving of network bandwidth and increasing the overall performance by allowing the ELD agent to process the power system data in the client machine itself. The proposed agent framework supports the asynchronous processing that makes ELD agent to fulfill a given task without the need to have a permanent connection from the client to a main container. The proposed architecture makes the economic dispatch application as more fault tolerant where network failure can influence only the migration of an ELD agent as the rest of the process is then done locally on the same node. A distributed environment has been set up using Java Agent Development Framework (JADE) to estimate and to monitor economic load dispatch
solutions for different sub-systems of an integrated power system. In this model the tie line power flow for each area is assumed to be constant and treated as load for each sub system. These power system clients are interconnected in a network where ELD agent is always on the move as shown in Figure 4.9.

Figure 4.9 Mobile agent model for Economic load dispatch

A client computer basically does the distributed power system monitoring through a GUI for every specific period of time and frequently exchanges data with the ELD agent. The ELD agent does the economic load dispatch computation and then distributes the results.
Chronologically the ELD mobile agent should start the process first in the main container (host1), so that it can take the initiative to set up a connection link. The mobile agent object obtains the necessary data from the container and responds back to them with the economic dispatch solutions. Then it migrates to the next host and serves the economic dispatch executable code. This total process of migration of ELD agent from one host to another has been made for every specific time interval. Transaction of ELD agent from different hosts takes place several times, so the possibilities of the occurrence of errors may be high in power system data communication. Hence it must be handled properly and it is taken care by the JADE distributed environment automatically.

4.3.1 Mobile Agent Flow Model

In the proposed mobile agent model, each neighboring power system is considered as a client container and ELD mobile agent is made to migrate from one container to another as shown in Figure 4.10. At first instance, ELD mobile agent m1 calls one of its environment’s (env1) methods requesting its migration to new environment env2 and the current environment creates a network connection with the new environment. Once the new environment accepts the network connection from the current environment, the current environment uses an ObjectOutputStream to serialize the mobile agent to a stream of bytes and sends the stream to the new environment.
At another instance the new environment calls the `readObject()` method of an `ObjectInputStream`, so it can reconstruct the ELD mobile agent from the byte stream. After this instance the new environment sends the acknowledgement to the current environment that it received the agent. Since the instance of the mobile agent in the new environment is not yet active, the current environment tells the instance of mobile agent to shutdown and it sends a confirmation to the new environment that the old instance of the agent has been shutdown. Once this shutdown status confirmation is received, the new environment allows the ELD mobile agent instance to be active. This collaboration of work makes the mobile agent to migrate to a new environment.

**Figure 4.10 ELD Mobile Agent Data flow**
4.3.2 Mobile Agent Paradigm

The major components of the mobile agent architecture are an agent manager, a security manager, an application manager and a directory manager as shown in Figure 4.11. All these major components mentioned have individual responsibilities to perform. The agent manager receives ELD agent for execution on the local host and also sends agents to the remote power system clients. The agent manager serializes the agent and its state before transportation of an agent to a power system client. It then passes the serialized form to its counterpart on the destination host. In a highly reliable architecture it actually passes the agent to the reliability manager, which ensures that the agent manager on the remote host receives the agent.

![Figure 4.11 ELD Mobile agent Paradigm](image-url)
Upon receipt of an agent, the agent manager reconstructs the agent object it references and then creates its execution context. The Security manager authenticates the agent before it is allowed to execute. Thereafter, the Java virtual machine automatically invokes the security manager to authorize any operations using system resources. ELD mobile agents may use directory manager to identify the location of a next neighboring host. An arriving mobile agent can access the resident servers such as database servers through this gateway. ELD Logic encapsulates the agent behavior and logic of economic dispatch application.

This architecture prescribes that an agent is a composite Java object that includes mobility, persistence and can communicate with other hosts. During its life cycle a mobile agent receives various kinds of events in response to its actions. For instance, if it moves to another host, a mobility event occurs just before and after the migration and corresponding call-back mechanism is invoked. In this way, each event gives to the agent the opportunity to determine how to react.

4.3.3 ELD Agent Life Cycle

An ELD mobile agent can be in one of several states, according to agent Platform Life Cycle in Foundation Intelligent for Intelligent Physical Agents (FIPA) specification; these are represented by some constants in Agent class in the JADE environment as shown in Figure 4.12.
The states are:

**AP_INITIATED**: The ELD agent object is built, but hasn't registered itself yet with the Agent Management System (AMS), has neither a name nor an address and cannot communicate with other agents.

**AP_ACTIVE**: The ELD agent object is registered with the AMS that has a regular name and address and can access all the various JADE features.
**AP SUSPENDED:** The ELD agent object is currently stopped. Its internal thread is suspended and no agent behavior is being executed.

**AP_WAITING:** The agent object is blocked, waiting for message. Its internal thread is sleeping on a Java monitor and will wake up when some condition is met (typically when a message arrives).

**AP_DELETED:** The ELD agent is definitely dead. The internal thread has terminated its execution and the Agent is no more registered with the AMS.

**AP_TRANSIT:** A mobile agent enters this state while it is migrating to the new location. The system continues to buffer messages that will then be sent to its new location.

**AP_GONE:** JADE internally uses this state when a mobile agent has migrated to a new location and has a stable state.

### 4.4 IMPLEMENTATION

Figure 4.13 shows the behavior of the ELD agent developed. The behavior class of the mobile agent uses two resources, in particular the counter and the counter enabled flag of the agent object. These two resources decide the time factor of the mobile agent that will reside in the power system client.
class ELDagentBehaviour extends SimpleBehaviour {
    ELDagentBehaviour(Agent a) {
        super(a);
    }
    public boolean done() {
        return false;
    }
    public void action() {
        if ( ((MobileAgent) myAgent).cntEnabled )
        {
            ((MobileAgent) myAgent).cnt++;
            String a1 = ((MobileAgent) myAgent).firstcall();
            System.out.println(a1);
            if( a1.endsWith("t") )
            {
                String eldresult = ((MobileAgent)myAgent).
                    calculateEconomicLoadDispatch(a1);
                System.out.println(eldresult);
                ((MobileAgent) myAgent).displayCounter(s);
            }
        }
        // Block the behavior
        block(10000);
        return;
    }
}

Figure 4.13 Behavior class of the ELD agent
Figure 4.14 shows the extension of class SimpleAchieveREInitiator in order to request to the Agent Management System the list of available power system clients in the locations where the ELD agent can move. Then, it displays results of economic load dispatch of the respective power system client through its GUI.

```java
public class GetAvailableLocationsBehaviour extends SimpleAchieveREInitiator {

    private ACLMessage request;

    public GetAvailableLocationsBehaviour(MobileAgent a) {

        // call the constructor of FipaRequestInitiatorBehaviour
        super(a, new ACLMessage(ACLMessage.REQUEST));
        request = (ACLMessage)getDataStore().get(REQUEST_KEY);

        // fills all parameters of the request ACLMessage
        request.clearAllReceiverQ;
        request.addReceiver(a.getAMS());
        request.setLanguage(FIPANames.ContentLanguage.FIPASLO);
        request.setOntology(MobilityOntology.NAME);
        request.setProtocol("fipa-request");

    }
}
```

**Figure 4.14 Mobile agent code for locating power system clients in the network**

The Mobile agent has to call the `getAMS()` method in order to find the next host address where it has to migrate.
Figure 4.15 shows the implementation of the economic load dispatch logic that resides in the *MobileAgent* class. In the *MobileAgent* class, the method `calculateEconomicLoadDispatch()` contains economic load dispatch logic and the method `setup()` is used to set the initial property of the ELD agent during execution.

```java
public class MobileAgent extends GuiAgent
{
    public void setup()
    {
        getContentManager().registerLanguage(new SLCodec(),
                FIPANames.ContentLanguage.FIPASLO);
        getContentManager().
                registerOntology(MobilityOntology.getInstance());
        gui = new MobileAgentGui(this);
        gui.setVisible(true);
        addBehaviour(new GetAvailableLocationsBehaviour(this));
        // incoming messages
        Behaviour b1 = new CounterBehaviour(this);
        addBehaviour(b1);
        Behaviour b2 = new ServeIncomingMessagesBehaviour(this);
        addBehaviour(b2);
    }

    public String calculateEconomicLoadDispatch(String s)
    {
        //Economic Load dispatch logic
    }

    protected void afterMove()
    {
        // Agent move to then next power system client
    }
}
```

**Figure 4.15 Implementation of an ELD agent**
In this agent implementation the \textit{afterMove()} method decides the agent property or functionality of an ELD agent while migration. The above distributed mobile agent model has been implemented in Windows NT based HP workstations connected in an Ethernet LAN. The results are shown in a GUI as given in Figure 4.16.

![Graphical User Interface](image)

**Figure 4.16 Economic load dispatch solution**

The above GUI shows the economic load dispatch solution for a specific 3-generator bus power system client. In this GUI, once start button event is triggered ELD agent will receive the power system data and starts the
computation of economic load dispatch solution. Once the economic dispatch solutions are displayed the "move" event has to be triggered by selecting the container or environment in the available locations window where the ELD mobile agent has to move. Using this approach, different power system clients can monitor continuous, updated and optimized economic load dispatch solutions at regular time intervals. An effective distributed agent model has been developed to monitor the economic load dispatch of multi-area power systems. Although, client-server architecture for economic load dispatch solution is very well established, the value of this study lies in that it emphasizes a unique methodology based on mobile agent model to serve a large number of clients in a distributed power system environment across various JADE platforms based on communication between virtual machines. A practical implementation of this approach suggested in this chapter was assessed based on 3, 6, 9 and 13 bus power systems. Accordingly the proposed model can be implemented for large power systems network spread over geographically apart.

4.5 CONCLUSION

A remoting architectural model has been presented in this chapter for on-line economic load dispatch in a distributed environment. The versatility of the proposed work is language independent based on common language runtime environment. The model is very flexible that provides the capability to extend power system application by adding custom components and different versions of economic load dispatch logic components that are serviced without affecting the power system client. Implementation of the proposed model and the steps for running the proposed application are discussed. A mobile agent based
on-line economic load dispatch model has also been presented. The presented agent model has been more effective in saving the network bandwidth and allows the ELD agent to process the power system data in the client machine itself. The proposed agent model makes the economic dispatch application as more fault tolerant where network failure can influence only the migration of an ELD agent as the rest of the process is then done locally on the same node, thus continuous service and monitoring of economic load dispatch solutions are achieved.