CHAPTER 3

ENHANCED DISTRIBUTED MODELS FOR POWER SYSTEM ANALYSIS

3.1 INTRODUCTION

The evolution of distributed computer architecture resulted in the birth of client-server architecture. In distributed computing, the client-server paradigm refers to a model of network applications where a server process is dedicated to managing access to some network service, while the client processes access the server to obtain a network service. Even though the client-server model provides a more cost effective means of computing, developing client-server programs is more expensive than traditional approaches with respect to power system applications. Distributed object computing overcomes the drawbacks with the monolithic and client-server applications and performs computations by a more flexible and cost effective approach. On-line power system analysis can be carried out in distributed environment using remote procedure calls or using sockets. Always there is a trade-off between the RMI API and Socket API while distributed applications have been deployed. The Socket API works closely with the operating system and hence has less execution overhead. RMI API provides the abstraction that eases the task of distributed application development. The choice of an appropriate paradigm and an appropriate API is a key decision in the design of a distributed application (Liu et al 2003).
3.2 RMI BASED LOAD FLOW MONITORING

The RMI system allows an object running in one virtual machine to invoke methods on an object running in another Java VM. RMI provides the mechanism by which the server and client communicates and pass information back and forth. RMI server creates remote services objects, makes references to them accessible and waits for clients to invoke methods on those remote objects. An RMI client gets a remote reference to one or more remote objects in the server and then invokes methods on them. An RMI based distributed application uses the registry to obtain a reference to a remote object. The RMI server calls the registry to associate a name with a remote object. The client looks up the remote object by its name in the server’s registry and then invokes a method on it.

This chapter outlines a new approach for load flow monitoring in a distributed environment. RMI based implementation of power system operation and control eliminates the overheads associated with conventional approaches. RMI uses built in security mechanism and hence the distributed load flow monitoring through an applet definitely secures the safety of the server as well as the power system data transfer.

In this present work, a distributed environment has been set up using RMI to estimate and to monitor load flow solutions for different sub-systems of an integrated power system. Each sub system has been considered as a power system client and hence multiple power system clients - single load flow server model is implemented. These power system clients are interconnected with a load flow server as shown in Figure 3.1. When there is a call for a method
located in a remote object, Figure 3.1 shows the flow of remote method invocation.

A client computer basically does the distributed power system monitoring through an applet for every specific period of time and frequently exchanges data with the server. The server does the load flow 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 load flow results at specific intervals of time. The total process has been automated by making the server to get the input data from each registered client at every specific period of time.

3.2.1 RMI Data flow Model

In the proposed model, each neighboring power system is considered as a client remote object. The power system client calls a method on an object that represents the remote object, which is called a stub. The stub contains a method for each of the methods in the remote object. Load flow stub always resides on the client's side and it packages the power system data into a block of bytes that can be communicated through the network. This process of marshalling presents the entire load flow data in a suitable format i.e., encoded byte array stream for transporting from one virtual machine to another. The load flow stub on the client's side builds from the information block that consists of an identifier of the remote object to be used, a description of the method to be called and the marshalled load flow data. When the load flow stub sends the information to the load flow server, a receiver object (skeleton) on the server side receives 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, the receiver object captures, marshals
the return value and sends the marshalled load flow results as packets on to a marshal stream and thus sends the load flow results to the stub. The stub unmarshals the return value and returns it to the original caller and the sequence of data flow is shown in Figure 3.2.

Figure 3.2 Invoking a load flow method on a remote object

In the proposed RMI based load flow model, the participating power system clients are notified by the remote load flow server when the specified time event occurs. With in the framework of the conventional RMI API it is not possible for the server to initiate a call to the client to transmit some information when the information becomes available or to obtain the necessary
data from the client, since a remote method call is unidirectional i.e., from the client to the server. One way to accomplish the transmission of such information is for each power system client to poll the remote load flow server object by repeatedly invoking a remote method `isLoadFlowStarted()`, until that method returns true. The following code segment has shown in Figure 3.3 implements the concept of polling.

```java
LoadFlowInterface lf1 = (LoadFlowInterface) Naming.lookup(http://127.0.0.1:8080/loadflow);
while( !lf1.isLoadFlowStarted() ) { ... ; }
//start load flow estimation
```

Figure 3.3 Implementation of polling in automated Load Flow Estimation

Polling is indeed a technique employed in many distributed applications. But it is very costly technique in terms of system resources, as each remote method invocation takes up a separate thread on the server host, along with the system resources that its execution entails. The present work proposes an efficient technique that leads to a mechanism of client callback where the power system clients are also treated as remote objects and are interacting with the remote load flow server object and vice versa. It allows each power system client to register itself with the remote load flow server object so that the server may initiate a remote method call to the power system client objects when the awaited time event occurs. The Figure 3.4 compares the two techniques: polling and callback.
With client callbacks, the remote method invocations are now two-way, or duplex from the client to the server, and vice versa. When the load flow server object makes a callback, the roles of the two processes are reversed. It becomes a client of the power system client object so that the load flow server object may initiate a remote method invocation to the client object. Figure 3.5 illustrates the architecture for the proposed callback RMI model for load flow monitoring. Compared with the architecture for basic RMI, two sets of proxies are now required. One set is required for the server remote interface, as in basic RMI architecture. The other set of proxies is for an additional interface, the client remote interface. The client remote interface provides a remote method that can be invoked by the server for the callback. Figure 3.5 illustrates that each client object registers with the server for callback and then will be notified subsequently whenever the specified time event occurs.
For the callback, the client must provide a remote method, `loadFlowData()` that allows the server to invoke it to obtain the load flow data at specific time intervals. It can do so in a manner similar to the remote methods provided by the remote load flow server object. The server object's remote interface is coded as shown in Figure 3.6. On the server side, a remote method needs to be provided to allow a client to register for callback.
public interface LoadFlowInt extends Remote
{
    public void register(PowerSystemClientInt client) throws RemoteException;
    public String loadFlowEstimation(String busData, String lineData) throws RemoteException;
}

Figure 3.6 Remote Load Flow Server Object's Interface

A reference to an object that implements the power system client remote interface is accepted as an argument. It is necessary for the load flow server to employ a data structure to maintain a list of registered client interface references. The implementation of the load flow server remote object is given in Figure 3.7. A Vector reference is used to maintain the registered list. Each call to register() method results in a power system client object’s reference being added to the vector.

In the conventional RMI approach, the object server provides a remote interface that declares the remote methods that an object client may invoke. To achieve inter-remote object communication, a similar remote interface will need to be provided by the object client. The client remote interface should contain at least one method to be invoked by the server for the callbacks. The power system client remote interface’s code is given in Figure 3.8
public class LoadFlowImpl extends UnicastRemoteObject implements LoadFlowInt, Runnable
{
    Vector clients;
    Thread mythread;
    public LoadFlowImpl() throws RemoteException
    {
        clients = new Vector();
        mythread = new Thread(this);
        mythread.start();
    }
    public void register(PowerSystemClientInt client) throws RemoteException
    {
        clients.addElement(client);
    }
    public String loadFlowEstimation(String busData, String lineData) throws RemoteException
    {
        // Load flow calculation
        public void run()
        {
            while(true)
            {
                for(int i=0; i< clients.size(); i++)
                {
                    PowerSystemClientInt psclient = (PowerSystemClientI nt)
                        clients.elementAt(i);
                    psclient.loadFlowData();
                }
            }
            try
            {
                Thread.sleep(60000);
            }
            catch(InterruptedException e) { }
        }
    }
}

Figure 3.7 Remote load flow server object implementation
public interface PowerSystemClientInt extends Remote
{
    public void loadFlowData() throws RemoteException;
}

Figure 3.8 Power System Client Object’s Remote Interface

The method loadFlowData() is to be invoked by the remote server object when it makes the callbacks. The client remote interface needs to be implemented in a class, called PowerSystemClientImpl and the implementation code segment is shown in Figure 3.9.

public class PowerSystemClientImpl extends JApplet implements PowerSystemClientInt
{
    LoadFlowInt client;
    public void init()
    {
        UnicastRemoteObject.exportObject(this);
        client=(LoadFlowInt)Naming.lookup("rmi://localhost:8000/loadflow");
        client.register(this);
    }
    public void loadFlowData() throws RemoteException
    {
        //code to retrieve bus data and line data;
        client.loadFlowEstimation(busData,lineData);
    }
}

Figure 3.9 Power system client implementation
In this proposed model the load flow server object invokes the method `loadFlowData()` to all the registered power system clients with it and this callback method in turn invokes the method `loadFlowEstimation()` in the load flow server. The power system client object when its execution started, registers itself with the load flow server by invoking the `register()` method on the load flow server object. As with the load flow server remote interface, the compiler `rmic` should be applied to the implementation of the client remote interface to generate the proxies needed at run time.

3.2.2 Steps for Building a remote load flow server object

A revised description of the step-by-step procedure for building an remote load flow server object is presented as follows:

- A new directory is created to store the code files for load flow server interface, its implementation and the server application
- The remote server interface is specified in LoadFlowInt.java
- The implementation of the above interface is stored in LoadFlowImpl.java
- The RMI compiler `rmic` is used to process the implementation class and to generate stub file and skeleton file for the remote object
- New versions of stub and skeleton files have to be generated each time there is a change in the load flow server interface
- A remote interface should be created for every power system client and this client remote interface should contain at least one method to be invoked by the server for the callbacks.
The proposed remote client interface has the method `loadFlowData()`

- The interface and implementation files have to be compiled to generate .class files
- A copy of the power system client remote interface stub file `PowerSystemClientImpl_Stub.class` is to be obtained
- The remote load flow server object has to be activated

### 3.2.3 Steps for developing the power system client object

The steps involved in deploying a power system client object are described as follows:

- A new directory is created to store the code files for client remote interface and its implementation
- The power system remote client interface is specified in `PowerSystemClientInt.java`
- The implementation of the above interface is stored in `PowerSystemClientImpl.java`
- The RMI compiler `rmic` is used to process the implementation class `PowerSystemClientImpl.class` and to generate the stub file, `PowerSystemClientImpl_Stub.class` and the skeleton file `PowerSystemClientImpl_Skel.class` for the remote object
- A copy of the load flow server remote interface class file has to be obtained
- A copy of the load flow server remote interface stub file `LoadFlowImpl_Stub.class` has to be obtained
- The client applet is activated

Figure 3.10 illustrates the files needed on the two sides (load flow server and power system client) for the proposed model that makes use of power system client callback and it also illustrates the hierarchical order of the files to be placed in the directory.

---

**Figure 3.10** File placements for an RMI based load flow monitoring application with power system client callback
3.2.4 Load flow server’s self registry service

RMI provides bootstrap registry service to locate remote server objects. The server program registers remote objects with the bootstrap registry service and the clients retrieve the stubs for those objects as shown in Figure 3.11.

![Figure 3.11 Registry Customization](image)

In the proposed method, the load flow server creates its own registry and it maintains the stubs for the remote objects by itself and hence the server
no longer needs to depend on the bootstrap registry service provided by the RMI protocol. The implementation of the load flow server with self registry is shown in Figure 3.12.

```java
public class LoadFlowServer {
    public static void main(String[] args) throws Exception {
        LoadFlowImpl lf = new LoadFlowImpl();
        Registry reg = LocateRegistry.createRegistry(8000);
        reg.rebind("loadflow", lf);
    }
}
```

**Figure 3.12 Load flow Server Implementation**

The server creates the self registry at port 8000 and the remote object reference has been bound with this registry.

### 3.2.5 Dynamic class loading

In RMI client-server architecture, clients can communicate with the remote object only when the server side stub is available with the client. The stub can be loaded on the client’s side dynamically by an external web server as shown in Figure 3.13.
The steps involved in downloading RMI stubs are as follows:

- The remote object’s codebase is specified by the remote object’s server by setting the `java.rmi.server.codebase` property. The RMI load flow server registers the remote server object, bound to a name, with the RMI registry.

- The power system client makes a request for a reference to a named remote object. The reference to the remote object’s stub instance is what the client will use to make remote method calls to the load flow server object.
- The RMI registry returns the stub instance reference to the requested client.

- The codebase which the power system client uses, is the URL that is annotated to the stub instance when the stub class is loaded by the registry.

- The class definition for the stub is downloaded to the client dynamically.

- Now the power system client has all the information that it needs to invoke remote method on the load flow server object. The stub instance acts as a proxy to the remote object that exists on the server.

Any changes in the interface of the server side results in the modification of the stub and it will be made available for clients by dynamic class loading through an external server.

### 3.2.6 RMI Based load flow monitoring algorithm

When a client’s remote object registers with load flow server’s remote object, the server uses the remote client reference to invoke its method to obtain the load flow data from that client and then provides the service through its methods. Both client and server objects are considered as remote objects and thus inter-remote object communication is achieved. The server object uses a single thread of control to distribute the load flow solution simultaneously to all the clients registered with it. The proposed model is dynamic which allows a new power system client to register with the load flow
server object at run-time and get serviced. A kind of multicasting has been achieved through this multi client-single server model. Load flow server and clients have to store in them, the necessary object codes required for load flow calculations. Stubs for both client and server must be kept at a common location like web server for distribution. Subsequently, the following steps are to be carried out:

- Start load flow server. Load flow server should invoke its own registry service.
- Start power system client by dynamically loading the server's stub from the common sharable location in a web server.
- 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 and it computes the load flow result and returns it to the client.
- Client obtains the result from the server through load flow stub and provides a view of the results through an applet.
- For every specific period of time, server automatically receives system data from the client, thereby providing an automatic load flow monitoring.

The above distributed algorithm has been implemented in Windows NT based HP workstations connected in an Ethernet LAN. The results are shown in a client applet as given in Figure 3.14.
Applet started.

Figure 3.14 Applet with load flow solution

The above applet shows the load flow solution for a specific 10-bus power system client. When each power system client applet is loaded, it registers with the load flow server, the server stub will be downloaded dynamically and through it, the client sends the request and receives the output. Using this approach, different power system clients can monitor continuous updated load flow solutions at regular time intervals.

Each power system client maintains the load flow data in its own database and hence a heterogeneous distributed database environment has been
created. The major advantage of holding the database on the client side is that the database connectivity overhead is brought to the lowest possible. The client remote object is tuned to provide load flow data from the database using a platform-independent JDBC architecture. In this proposed model two types of JDBC drivers have been implemented to operate heterogeneous client databases. The first one is the usual JDBC-ODBC bridge driver. The second one is the customized JDBC net driver, which is to be loaded at the client’s end that communicates directly with the DBMS protocol of the load flow server. Each power system client can also monitor the load flow of other power system clients by entering the identification of the registered client. The Figure 3.15 shows that the power station B monitors the load flow solution of power station A.

Figure 3.15 Power System Load flow monitoring Applet
3.3 A DISTRIBUTED MODEL FOR MULTI-AREA POWER SYSTEMS ON-LINE DYNAMIC SECURITY ANALYSIS

In the past when power systems were smaller and less complicated, informal methods of security analysis and control were performed by system operators based upon their experience and knowledge of the system. Modern power systems are quite large and more extensively interconnected, making the task of security analysis and control difficult for the system operator. Moreover, during times of system difficulty the system operator is often pre-occupied to the point that he may not be able to properly interpret worsening conditions or the development of new problems. As the number of contingencies increases, the solution or execution time increases to a point which may be so long as to be cumbersome or meaningless for real-time dispatcher response. Under these circumstances, it becomes desirable to develop a contingency selection method which would select only those cases considered severe requiring more detailed analysis.

There is a large saving in computational effort if only the relevant contingencies called 'critical' contingencies, which result in equipment violations. This process of selection of critical contingency cases is often referred to as 'Automatic Contingency Selection'. The contingency selection algorithm is used to prepare the contingency list. This list is then ranked with the most severe contingency at the top of the list and proceeding down the list, stopping when the severity goes below a threshold. This contingency list mainly depends upon the load levels, which will be changing too often. Thus the selection algorithm is to be executed periodically, to update the list. It is anticipated that it will be performed only two or three times a day or upon
analysis task that will be executed every two or three minutes utilizing the greatly reduced list produced by the contingency selection procedure. The automatic contingency selection algorithms developed in this chapter are based on performance indices defined separately to reflect line overloads, unacceptable voltage levels and reactive power capabilities of generators. Thus the separate rank list can be obtained for any of these three options. The lists of contingencies are prepared starting from the higher one. Detailed studies are then carried out for the identified critical contingencies obtained by the ranking procedure.

The key aspect of the new development in power system on-line control is the enhancement of the security of the power system in order to maintain a high reliability of electric power supply. Security of the power system requires the proper integration of both automatic and manual control functions. Both the steady state and dynamic state emergency conditions of a power system operating problem have to be characterized by keeping the system operating optimally based on voltage, real and reactive power. During the normal state in order to forestall an emergency and reassure that all loads to be satisfied, it is essential to conduct security analysis to know the system is vulnerable to electrical disturbance. The first function of the security analysis is to determine whether the normal system is secure or insecure by contingency evaluation and the second function is to determine the preventive action should be taken when the system is insecure. As the number of contingencies increases for multi-area power systems, contingency evaluation by conventional client - server architecture is complicated. In the conventional client - server applications for contingency security analysis, it is assumed that the information required for monitoring and controlling of power systems is
centrally available and the server has to be loaded every time for each client’s request and the time taken to deliver the critical contingencies also comparatively high.

In this present work, a distributed environment has been set up using RMI to estimate and to monitor contingency evaluation for different sub-systems of an integrated power system. The power system clients are interconnected with a contingency server as shown in Figure 3.16.

Figure 3.16 RMI based Client-Server Architecture for contingency analysis
The power system client basically does the power system security analysis through an applet, for every specific period of time by frequently exchanges the system data with the server. The server does the contingency ranking and then distributes the results. The above distributed algorithm has been implemented in Windows NT based HP workstations connected in an Ethernet LAN. The results are shown in a client applet as given in Figure 3.17.

![Applet Viewer: contourlot.png](image)

**Figure 3.17  Applet with contingency ranking with Performance Index**

The above applet shows the contingency ranking for a specific 8-bus power system client and the load flow result with a line is removed. When each power system client applet is loaded, it registers with the contingency server,
the server stub will be downloaded dynamically and through it, the client sends the request and receives the output. Using this approach, different power system clients can monitor continuous updated critical contingencies at regular time intervals.

3.4 RMI BASED DISTRIBUTED MODEL FOR ON-LINE POWER SYSTEM ECONOMIC LOAD DISPATCH

The Economic dispatch problem of power systems is to determine the optimal combination of power outputs for all generating units, which minimizes the total fuel cost while satisfying the constraints. The proposed RMI based distributed model has been extended to solve the economic load dispatch problems. A distributed environment has been set up using RMI to estimate and to monitor economic load dispatch solutions for different sub-systems of an integrated power system. An economic load dispatch server (ELD server) object has been implemented and is used in the place of load flow server object in the previously stated contingency and load flow models. In this model the tie line power flow for each area is assumed to be constant and it is treated as load for each sub system. These power system clients are interconnected with the ELD server. The economic load dispatch results are shown in a client applet as given in Figure 3.18.
Figure 3.18 Applet with economic load dispatch solution

The applet shows the economic load dispatch solution for a specific 3-generator bus power system client.

3.5 RMI BASED GROUP COMMUNICATION MODEL FOR ON-LINE POWER SYSTEM ANALYSIS

In the conventional client-server architecture for load flow monitoring, the data communication between the load flow server and power
system client is through TCP/IP communication, which is unicast. In the proposed group communication model, a power system client can dispatch load flow data to the load flow server, load flow results to the contingency server, the data required for estimating economic load dispatch to the economic load dispatch server and can obtain the respective solutions from each server. Figure 3.19 illustrates the group communication model for on-line power system analysis in a distributed environment.

Figure 3.19 RMI based group communication model for on-line power system analysis
Each remote power system client object can automate the entire process of power system analysis by generating the sequence of operations in a command list. The command list consists of a sequence of IP addresses of the remote server objects and each IP address is followed by the command that represents the action taken by the corresponding remote server object. The sequence of power system operations is controlled by this command list. The format of the command list is shown below:

Command List = {IP address 1, LoadFlow, IP address 2, Contigency, IPaddress 3, ELD, ..........., IP address n, StateEstimation};

The sequence of power system operations in the command list may be in any order as per the power system client requirements. The power system client transmits the power system data to the respective server as a single string. The string consists of command list followed by a power system data. In this proposed model the power system client sends the load flow request by invoking loadFlowEstimation() in the LoadFlowServer. The StringTokenizer class is used to tokenize separately the load flow data and the IP addresses in the command list from the load flow request packet. If the number of tokens exceeds two, then the next server IP address has to be found, because power system client has to access the next contingency server or Economic Load Dispatch server to find the critical contingency list or economic load dispatch result. Load flow server computes the load flow for the data given by the power system client and it also removes the first IP address in the command list which indicates its own IP address. The updated command list and the load flow results have been sent back to the power system client. Based on the IP address list the power system client can invoke the next server. If the command
list vector received by the power system client is null, it indicates that the on-line power system analysis has been completed successfully by that particular power system client.

3.6 RMI BASED ACTIVE NETWORKING MODEL FOR ON-LINE DYNAMIC SECURITY ANALYSIS

The major drawback in the on-line dynamic security analysis through group communication model is that every time contingency server has to depend on load flow server to deliver results for each line outage through power system client. There is no straight communication between the load flow server and the contingency server. An active networking model is proposed as shown in Figure 3.20 through which the servers can communicate with each other.

![Figure 3.20 Active Network model for on-line dynamic security analysis](image-url)

Figure 3.20 Active Network model for on-line dynamic security analysis
In the proposed model the power system client sends the load flow data to the Load flow server and finally it will obtain the critical contingency ranking list from the remote contingency server. Once the load flow data is sent to the load flow server it computes the load flow results for every line outage and sends the results to the contingency server. The active network model has been developed such that the load flow server becomes the client remote object whenever there is a load flow request from contingency server. This inter-server communication between load flow server and contingency server continues until the critical contingencies are listed based on performance index. Finally the critical contingency list due to bus voltages, real power and reactive power is sent to the power system client. In this proposed model the remote methods such as \textit{loadFlowEstimation()} and \textit{cotingency()} are declared as synchronized to avoid the change in order while more than one power system client is processing the on-line dynamic security analysis.

\textbf{3.7 COMPONENT MODELS FOR ON-LINE POWER SYSTEM ANALYSIS}

Enterprise load flow component models are pluggable, reusable and can simplify the complexity in the areas of synchronization, scalability, load flow monitoring integrity, networking and distributed object frameworks. A load flow bean once developed can be deployed on multiple platforms without recompilation or source code modification.

In this proposed model, each power system client can access the remote load flow Enterprise Java Bean (EJB) server through the servlets using data object serialization. The load flow server in turn computes and
disseminates load flow solutions to all the power system clients simultaneously for every specific period of time based on the client's requirement. The various entities of the proposed EJB model are the load flow EJB server, the load flow EJB container that runs within the server, home objects, remote EJB Objects, load flow bean that executes within EJB container, power system clients and JNDI Services. The relationship between the above entities of the proposed EJB model is shown in Figure 3.21.

Figure 3.21 Component model for Load flow Monitoring

The load flow EJB server provides an organized framework or execution environment in which EJB container can run. It makes available
system services for multiprocessing, load balancing and device access for EJB containers. The J2EE platform enables a multi-tiered, distributed application model, the ability to re-use components, a unified security model, and flexible transaction control. Power system clients simultaneously access the load flow EJB server through Java Naming and Directory Interface (JNDI) naming service. Based on the client's requirement, the server communicates with the remote client, fetches the present load flow data, computes load flow solution and provides the result to that specific client. The process is simultaneously done for every registered client by generating a separate thread of control. The purpose of loading the server with load flow computational skill is that any further modification to the computation methodology would reflect appropriate results for all the remote clients.

The load flow bean component lies inside the load flow EJB container. The load flow EJB container provides services such as load flow calculation management, versioning, scalability, mobility, persistence, and security to the components it contains. Since the EJB container handles all of these functions, development of load flow component is made easy and EJB container contains the load flow bean, Home and Remote interfaces. The load flow bean executes within a load flow EJB container, which in turn executes within an EJB Server. A load flow EJB component is the type of EJB class, which comprises the load flow computation logic. All the other classes in the EJB system support either client access or provide services to EJB component classes. In this proposed architecture, the load flow bean is a stateless session bean. A stateless load flow bean does not maintain a conversational state for a particular power system client. When a power system client invokes the method of a load flow bean, the bean's instance variable may remain in a particular state, but only for the duration of the invocation. When the operation
of the method is over, the state is no longer retained. Stateless session beans can support multiple clients and it can offer better scalability for load flow monitoring application for large power system clients.

Load flow EJB component which is the Home Interface, defines the methods for creating, initializing and destroying the instances of the server. The home interface is a contract between a load flow EJB component class and its container, which defines construction, destruction, and looks up the EJB instances. A load flow EJB home interface extends the interface `javax.ejb.EJBHome`, which defines base-level functionality for a home interface and all methods in this interface must be RMI-compatible. The Remote Interface lists the load flow method available in the bean. The EJB object is the client's view of the enterprise bean and implements the remote interface. While the load flow bean defines the remote interface, the container generates the implementation code for the corresponding EJB object. Each time the power system client invokes the EJB object's method while the EJB container handles the request before delegating it to the load flow bean. Power system clients locate the specific load flow EJB container that contains the load flow bean through the JNDI service. They make use of the EJB container to invoke the load flow bean and get a reference to an EJBObject instance. When the client invokes a method, the EJBObject instance receives the request and delegates it to the corresponding bean instance and also provides necessary wrapping functionality.

In this proposed method, load flow monitoring by each client is achieved through an applet - servlet - EJB communication model for every specific period of time. The applet to servlet communication is enabled via HTTP tunneling. The power system client transforms the load flow data into a
stream of bytes using ByteArrayOutputStream that has been sent as a load flow request to the server side application which transmits the same to the load flow bean. The load flow results have been constituted as response object and sent back to each power system client at specific intervals of time.

The power system client applet opens a URL connection to the server side application, passing it by name and the port number of the remote host which can upload the load flow data to the EJB load flow server. JNDI adds value to load flow bean deployment by providing standard interface for power system clients. Naming service in JNDI is the entity that associates names with objects and it provides a facility to find an object based on its name. JNDI is a unified system to access all sorts of directory service information such as security credentials, machine configurations and network address of the power system clients. The greatest use of JNDI service is to locate load flow bean’s home object. The reference of the load flow home object is acquired by declaratively specifying the environment properties files or system files which detail the JNDI service provider used in the load flow bean deployment. It uses the environment properties employed in creating the initial context factory to look up the load flow object stored in the directory.

3.7.1 EJB data flow model

Power system clients use the JNDI service to lookup load flow objects over a network. A remote power system client accesses the load flow bean through its remote and home interfaces. When the power system client performs a JNDI lookup for a home object, the EJB container might use JNDI to return a remote stub. The remote stub is a proxy for the load flow home
object, which is located elsewhere in the network and once the power system client has a stub, it can invoke a load flow method on the home object through the remote stub object. The EJB object that implements the remote and home interfaces are accessible from a client through the standard RMI APIs. It communicates with the remote EJB container and then with the load flow bean thus requesting that the load flow method as shown in Figure 3.22. The load flow EJB container executes the load flow bean and sends the load flow solutions back to each power system client via a servlet at specific intervals.

Figure 3.22 Invoking a load flow method on the remote EJB Server
3.7.2 Load flow Bean Life Cycle

The life cycle of a load flow bean instance is described as follows:

- A stateless load flow bean instance’s life starts when the container invokes `newInstance()` on the load flow bean class to create a new instance and the container calls `setSessionContext()` followed by `ejbCreate()` on the instance as shown in Figure 3.23. The container can perform the instance creation at any time and there is no relationship to the client’s invocation of the `create()` method.

- The session bean instance is now ready to delegate a load flow method call from any power system client.

- When the load flow bean is no longer needed, the container invokes the `ejbRemove()` method. This ends the life of the stateless load flow bean instance.

Figure 3.23 Life cycle of EJB load flow Bean
3.7.3 Deployment procedure of component based load flow monitoring

In order to deploy the load flow bean component into the load flow server, the following steps are to be followed:

- Start the J2EE deploytool window and select the new application.
- Choose the corresponding enterprise archive file and type the application display name.
- Start the New Enterprise wizard to package the load flow bean and type the JAR display name.
- Add the `loadflowint.class` and `loadflowHome.class` and `loadflowEJB.class` to JAR dialog box.
- In the General dialog box, choose the bean type as stateless session bean and choose appropriate interfaces in the Enterprise Bean class and enter the name of the enterprise bean as load flow bean.
- Open the deploy wizard and give the full path of client’s jar file name which contains the stub classes and it will enable remote access to the load flow bean.
- Enter the JNDI name and WAR context root and deploy the load flow bean.
A complete component model for load flow monitoring has been implemented in Windows NT based HP workstations connected in an Ethernet LAN. The results are shown in a client applet as given in Figure 3.24.

![Figure 3.24 Power System Load flow solving Applet](image)

The above applet shows the load flow solution for a specific 10-bus power system client. When each power system client applet is loaded, it invokes the servlet via http tunneling and in turn the servlet accesses the load flow bean by its JNDI name and the Web Context root. The EJB container runs the load flow bean automatically and the load flow solution is calculated and result is sent back to the respective power system client.
3.8 COMPONENT MODEL FOR ON-LINE DYNAMIC SECURITY ANALYSIS

In this proposed model, each power system client can access the remote contingency bean for on-line dynamic security analysis. The contingency server in turn computes and disseminates contingency ranking to all the power system clients simultaneously for every specific period of time based on client's requirement. The load flow and contingency bean components are deployed within an EJB container. The relationship between the components of the proposed EJB model is shown in Figure 3.25.

Power system clients simultaneously access the contingency server through JNDI naming service. Based on the client's requirement, the server communicates with the remote client, fetches the present contingency data and computes the load flow by considering the specified line outage. The load flow solutions are used by the server to compute the performance index and provide the contingency ranking list to the respective clients. The process is simultaneously done for every registered client by generating a separate thread of control. A contingency bean component is the type of EJB class which comprises the contingency ranking procedure. All the other classes in the EJB system support either client access or provide services to EJB component classes. In this proposed architecture, load flow bean and contingency bean are stateless session beans thus offering better scalability for dynamic security analysis for large power system clients.
Figure 3.25 Component model for dynamic security analysis of multi-area power systems

Contingency EJB container executes the load flow bean and sends the load flow solution to the contingency bean, which in turn computes the performance index for each contingency. The data flow model for the interaction between Load flow bean and Contingency bean is shown in Figure 3.26. Critical contingencies are ranked based on their performance index the rank list is sent to each power system client via servlet at specific intervals.
Complete component model for on-line dynamic security has been implemented in Windows NT based HP workstations connected in an Ethernet LAN. The results are shown in a client applet as given in Figure 3.27.
The above applet shows the contingency ranking for a specific 8-bus power system client and the load flow result with a line is removed. When each power system client applet is loaded, it invokes the servlet via http tunneling and in turn the servlet accesses the load flow bean by its JNDI name and Web Context root. EJB container runs the load flow bean automatically and sends the load flow solution as input to the contingency bean. In turn contingency bean calculates the performance index based on the voltage, real and reactive power. Finally critical contingency list has been sent back to respective power system client and hence the power system clients can monitor critical
contingencies at regular time intervals. Based on the proposed work, a component model for on-line economic load dispatch has also been developed and implemented.

3.9 LOCATION INDEPENDENT DISTRIBUTED MODEL FOR ON-LINE POWER SYSTEM ANALYSIS

It is aimed at to develop location transparent distributed model for on-line power system analysis. The location transparency is the key feature of Common Object Request Broker Architecture (CORBA). Location transparency of the proposed model is the ability to access and invoke operations on the CORBA server object without needing to know where the power system object resides. Developed distributed model also provides language transparency that facilitates the implementation of the power system logic in any programming language.

3.9.1 Location transparent model for load flow monitoring

In this proposed model, the power system client can access the remote load flow server through the CORBA server using Internet Inter ORB Protocol (IIOP) as shown in Figure 3.28. In this model the power system client is represented as a Java applet and it can be downloaded in the client machine. The power system client applet is designed in such a way that it maintains the previous state until it receives the converged load flow results from the load flow server for a given load flow data.
Figure 3.28 CORBA based model for load flow monitoring

Figure 3.28 illustrates the simplest scenario where a power system client interacts with the load flow server through object request brokers (ORB). The power system client and the load flow server are both implemented in JVM. The power system client communicates with the ORB in order to convey a request for an operation invocation to the load flow server, which receives the power system data and then sends the load flow results via the ORB back to the power system client. The interfaces of these components are defined by the CORBA standard and by the application specific IDL.

3.9.2 CORBA data flow model

This Figure 3.29 shows a more concrete view of how the ORB performs the task of conveying an invocation from power system client to the load flow server. The IDL compiler generates a number of Java classes known
as stub classes for the client and skeleton classes for the load flow server. The role of the stub class is to provide code for proxy object on which power system clients invoke load flow method. The proxy object method implementations on power system client side invoke operations on the load flow servant, which may be located remotely. If the servant is at a remote location the proxy marshals the power system data and transmits the invocation request. It takes the name of the operation and the types and the values of its arguments from language-dependent data structures and places them into a linear representation suitable for transmitting across a network.

![Figure 3.29 CORBA based data flow model](image)

The resulting marshaled form of the request is sent to the load flow servant using the particular ORB's infrastructure. In this proposed work, the infrastructure involves a network transport mechanism and additional mechanism to locate the load flow servant and perhaps to activate the CORBA
server programs that hosts the servant. The server side skeleton code provides the glue between a load flow server object implementation, a CORBA server, and the ORB, in particular the object adapter. The CORBA specification leaves many of the interfaces between the ORB core, object adapter, and load flow program partially or totally unspecified. For this reason different ORBs have different mechanisms to activate load flow server and for use by object adapters to inform the ORB that their objects are ready to receive invocation requests. After receiving a request, the ORB consults the object adaptor to find the load flow server that is going to execute the operation. The skeleton of the load flow server class implements the mechanism by which invocation requests coming into a load flow server can be unmarshaled and directed to the load flow method of a servant. The steps involved in the development of CORBA based distributed load flow monitoring application are detailed as follows:

- Write IDL that describes the LoadFlowInterface to the load flow server object that will be implemented. Compile the LoadFlow IDL file.
- This produces the stub and skeleton code that provides location transparency. That is it will cooperate with the ORB library to convert an object reference into a network connection to a remote load flow server and then marshal the power system data as arguments to an operation on the object reference, convey them to the loadflow method in the server object, execute the method and return the load flow results. Compile the .java files, including the stubs and skeletons.
• Identify the IDL compiler generated interfaces and the classes that need to be used in order to invoke or implement load flow monitoring in a distributed environment method.

• The ORB has to be initialized and it has to inform about the load flow server remote objects created. Compile all the generated code and run the distributed Load flow monitoring application.

The location transparent model developed for load flow monitoring has been extended for on-line dynamic security analysis and economic load dispatch problems.

3.9.3 Implementation

All CORBA objects support an IDL interface; the IDL interface defines an object type. An interface can inherit from one or more other interfaces. The IDL file functionality is the CORBA language-independent analog to a C++ header file. IDL is mapped into each programming language to provide access to object interfaces from that language. With Java IDL, these interfaces can be translated to Java using the idltojava compiler. For each IDL interface, idltojava generates a Java interface and the other .java files needed, including a client stub and a server skeleton. The IDL interface for the load flow estimation is shown in Figure 3.30.
module LoadflowApp
{
    interface Loadflow
    {
        String loadFlowEstimation( String );
    }
};

Figure 3.30 IDL interface for the Load flow estimation

The IDL interface is complied and this process generates five files in a LoadflowApp sub-directory:

_LoadflowImplBase.java: is an abstract class providing basic CORBA functionality for the load flow server. It implements the Loadflow.java interface.

_LoadflowStub.java: is the client stub, providing CORBA functionality for the power system client. Loadflow.java interface contains the Java version of IDL interface. It contains the method loadflowEstimation().

Loadflow.java: is the interface extends org.omg.CORBA.Object, providing standard CORBA object functionality as well.

LoadflowHelper.java: class provides auxiliary functionality, notably the narrow() method required to cast CORBA object references to their proper types.
**LoadflowHolder.java** is a class holds a public instance member of type Loadflow interface. It provides operations for out and inout arguments, which CORBA has but which do not map easily to Java's semantics.

The Java IDL Transient Name service is an object server provided with Java IDL. The Name server has been started using `tnameserv` at the command line prompt. This object server conforms to the standard object implementation and invocation techniques. The Name Server stores load flow server object references by name in a tree structure similar to a file directory. A power system client may lookup or resolves object reference by its name.

The load flow server consists of two classes, the load flow servant and the server. The servant, `LoadflowServant`, is the implementation of the Loadflow IDL interface; each Loadflow instance is implemented by a `LoadflowServant` instance. The servant is a subclass of `_LoadflowImplBase`, which is generated by the `idltojava` compiler. The servant contains `loadflowEstimation()` method for each IDL operation. Servant methods are just like ordinary Java methods; the extra code to deal with the ORB, with marshaling arguments and results, and so on, is provided by the server and the stubs. The Loadflow server class has the `main()` method which creates an ORB instance and creates a servant instance and intimates the ORB about it as shown in Figure 3.31. Load flow server CORBA object’s reference is obtained from a naming context to which a new CORBA server object is registered. The new servant object is registered in the naming context using the name "loadflow".
import LoadflowApp.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

class LoadflowServant extends _LoadflowImplBase
{
    public String loadFlowEstimation( String pdata)
    {
        // Load flow computation logic
    }
}

class LoadflowServer
{
    public static void main(String args[])
    {
        // create and initialize the ORB
        ORB orb = ORB.init(args, null);

        // create load flow servant and register it with the ORB
        LoadflowServant ref = new LoadflowServant();
        orb.connect(Ref);

        // get the root naming context
        org.omg.CORBA.Object objRef =
            orb.resolve_initial_references("NameService");
        NamingContext ncRef = NamingContextHelper.narrow(objRef);

        // bind the Object Reference in Naming
        NameComponent nc = new NameComponent("loadflow", "");
        NameComponent path[] = {nc};
        ncRef.rebind(path, loadflow);

        java.lang.Object sync = new java.lang.Object();
        synchronized (sync) {
            sync.wait();
        }
    }
}

Figure 3.31 Implementation of the Load flow server
Power system client code is linked with idltojava-generated .java files and the ORB library. It will create CORBA objects via the published factory interfaces that the server provides. Since a CORBA object may be shared by many clients around a network, only the object server is in a position to know when the object has become garbage. The client code's only way of issuing method requests on a CORBA object is via the load flow server object's object reference as shown in Figure 3.32.

```java
import LoadflowApp.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;
public class Loadflowclient {
    public static void main(String args[]) {
        // create and initialize the ORB
        ORB orb = ORB.init(args, null);
        // get the root naming context
        org.omg.CORBA.Object objRef = orb.resolve_initial_references("NameService");
        NamingContext ncRef = NamingContextHelper.narrow(objRef);
        // resolve the Object Reference in Naming
        NameComponent nc = new NameComponent("loadflow","");
        NameComponent path[] = {nc};
        loadflow Ref = loadflowHelper.narrow(ncRef.resolve(path));
        // call the load flow server object
        String Ifresult = Ref.loadFlowEstimation( );
        System.out.println(Iffesult);
    }
}
```

Figure 3.32 Power system client implementation for location transparency model
The object reference is an opaque structure which identifies a CORBA object's host machine, the port on which the host server is listening for requests, and a pointer to the specific object in the process. Because Java IDL supports only transient objects, this object reference becomes invalid if the load flow server process has stopped and restarted. Power system clients typically obtain object references from the name service. Once an object reference is obtained, the power system client must narrow it to the appropriate type and can invoke the `loadFlowEstimation()` method. The load flow results are monitored at regular intervals.

3.10 PERFORMANCE ANALYSIS

The core power system operations like load flow monitoring, economic load dispatch and on-line dynamic security analysis are being carried out using distributed technologies using RMI that provides platform independency and RMI-IIOP/CORBA that supports location independency. Components have been deployed for these core power system operations to ensure modularity and reusability. The major factor that influences the performance of the proposed models is the round trip time (RTT) that includes the convergence time. The round trip time measures the time needed from the point when the power system client initiates a method invocation to the point when the client receives the results. The round trip time is measured for all the power system clients that invoked the load flow method simultaneously without any delay. The performance analysis of the proposed distributed models has been carried out with respect to load flow monitoring and the variations of round trip time with respect to the number of clients are shown in Figure 3.33. The round trip time measures for a specific power system client are shown in Table 3.1.
Table 3.1 Performance analysis of the proposed distributed models

<table>
<thead>
<tr>
<th>Load flow monitoring</th>
<th>RTT in milli seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>15 Bus</td>
</tr>
<tr>
<td>Client/Server Model</td>
<td>68</td>
</tr>
<tr>
<td>RMI Model</td>
<td>45</td>
</tr>
<tr>
<td>Component Model</td>
<td>52</td>
</tr>
<tr>
<td>CORBA Model</td>
<td>71</td>
</tr>
</tbody>
</table>

Figure 3.33 RTT vs. No of Clients (Load flow monitoring)
The time taken to invoke the load flow method and to return the results increases linearly as the number of clients gets increased in both RMI and component models. The graph is plotted between the round trip time calculated and the number of clients registered with the server at a specific interval of time. From the above graph, it is found that RMI model performs better than the conventional client-server architecture and the component model. Similarly the performance analysis of the proposed distributed models has been carried out with respect to on-line dynamic security analysis and the variations of round trip time with respect to the number of clients are shown in Figure 3.34.

The round trip time is calculated after estimating the performance index of all line contingencies and these measures are tabulated in Table 3.2. Although the round trip time taken for contingency analysis using component model is higher compared to that of the RMI model, it provides a definite advantage of reusability by developing the load flow and the contingency components which interact with each other.

Table 3.2 Round trip time measures – on line dynamic security analysis

<table>
<thead>
<tr>
<th>Dynamic Security Analysis</th>
<th>RTT in milli seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8 Bus</td>
</tr>
<tr>
<td>Client/Server Model</td>
<td>290</td>
</tr>
<tr>
<td>RMI Model</td>
<td>267</td>
</tr>
<tr>
<td>Component Model</td>
<td>274</td>
</tr>
<tr>
<td>CORBA Model</td>
<td>294</td>
</tr>
</tbody>
</table>
The performance analysis of the on-line economic load dispatch models in a distributed environment is shown in Figure 3.35. Table 3.3 represents the round trip time measures for the economic load dispatch as estimated using the proposed distributed models.

Table 3.3  Round trip time measures for economic load dispatch

<table>
<thead>
<tr>
<th>Economic Load Dispatch</th>
<th>RTT in milli seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>9 Bus</td>
</tr>
<tr>
<td>Client/Server Model</td>
<td>65</td>
</tr>
<tr>
<td>RMI Model</td>
<td>53</td>
</tr>
<tr>
<td>Component Model</td>
<td>58</td>
</tr>
<tr>
<td>CORBA Model</td>
<td>71</td>
</tr>
</tbody>
</table>
The results show that the RMI model performs better than the conventional client/server architecture and the component model in all cases.

3.11 CONCLUSION

Distributed models for on-line power system analysis have been implemented in HP workstations connected in an Ethernet LAN. Although the client/server architecture for on-line power system analysis is well established, this thesis emphasizes a unique methodology based on RMI, RMI-IIOP and CORBA for the implementation in a distributed environment. An RMI based model for load flow monitoring in a distributed environment had been implemented and discussed in detail. A customized registry is created to bind the remote load flow server object and hence the resource consumed by the default RMI registry is released. A dynamic stub loading procedure is adopted
to enable the power system client to obtain the remote server object’s reference. This is achieved by placing the proxy of the remote load flow server object in a sharable location of a web server. A callback mechanism has been developed to automate the load flow monitoring at specific intervals of time. RMI based group communication and active network models have been designed and implemented to enhance the scalability of power system applications in a distributed environment. All these models deviate from the default behavior of RMI protocol which is always supporting unicast and hence inter-remote object communication has been achieved and thus reducing the computational time in solving power system applications in distributed environment. A component model for on-line power system analysis has also been proposed and implemented that increases the reusability and scalability of the system. A location transparent model has been developed using CORBA with RMI-IIOP for load flow monitoring and the idea is extended to solve on-line economic load dispatch and dynamic security analysis. The performance analysis of the proposed distributed models has been carried out with respect to load flow monitoring, dynamic security analysis and economic load dispatch. The variations of round trip time with respect to the number of clients are plotted as graph. RMI based model with callback mechanism performs better than all other models. CORBA based model consumes more RTT because every time the power system client has to search the CORBA service provider for the load flow service in the network. Both platform and location independent models for solving power system problems have been discussed in this chapter, whereas a language independent model and an agent model for providing continuous service in solving power system applications have been discussed in the next chapter.