M.Tech Dissertation

On

Design and Implementation of a Service Oriented ARC Framework for C#/.NET over LAN and Internet

Submitted in partial fulfillment of requirements
for the degree of
Master of Technology

By

T. Vamsi Kalyan.
Roll No: 01305605

Under the guidance of

Prof. Rushikesh K. Joshi.

Department of Computer Science and Engineering

Indian Institute of Technology, Bombay

Mumbai, 400076

Dissertation Approval Sheet

This is to certify that the dissertation titled
Design and Implementation of a Service Oriented ARC Framework for C#/.NET over LAN and Internet
By
T.Vamsi Kalyan.
(01305605)
is approved for the degree of **Master of Technology**.

Prof. Rushikesh K. Joshi.
(Guide)

Prof. Pushpak Bhattacharya
(Internal Examiner)

Mr. David D'lima
(External Examiner)

Prof. S. A. Khaparde
(Chairperson)

Date :

Acknowledgment

I express my sincere gratitude toward my guide **Prof. Rushikesh K. Joshi** for his constant help, encouragement and inspiration throughout the project work.

T.Vamsi Kalyan.
IIT Bombay
November 21, 2003

Abstract

Anonymous Remote Computing (ARC) for C# over .NET is a service oriented framework to support development of parallel and distributed programs in presence of mobility. The report provides a detailed description of the framework and discusses an associated applications development process. The basic design features of this framework are services for parallelism and distribution at object level. These include distribution of ARC objects, support for dynamic load measurement, fault tolerance and dynamic leave and join services for participating machines. In addition to these basic services, the framework is extended to support multiple object hopping and object accessibility in presence of mobility. The design and implementation of both the higher level and the kernel level in the ARC framework are discussed in detail. ARC framework is extended to work over the Internet by keeping software reuse as an objective. As a solution approach, webservices layer was added to the ARC framework architecture to use ASP.NET XML webservices infrastructure for communication between machines on the Internet. Some components of LAN version of ARC software were modified to address the differences between LAN and the Internet environments. Design and implementation of ARC kernel are both based on object-oriented technology. UML is chosen as a modeling language and implementation was carried out in C#. The report also includes deployment modeling of ARC, and some examples of distributed application programs written on top of the framework that were implemented on a cluster of Windows workstations.

Contents

List of Figures

Chapter 1

Introduction

Anonymous Remote Computing(ARC) [11] is a framework to support development of parallel programs over a cluster of workstations in presence of heterogeneity, load and failures. In an ARC environment, participating nodes may join and leave the system dynamically. In [10], a design and implementation of a service-oriented ARC kernel over a LINUX cluster has been discussed. Services are modeled through RPC based protocols. Though the modeling of this kernel was done through object modeling techniques, the implementation is procedural. Interfaces identified during modeling are translated into C-based RPC services during implementation.

In this report, an ARC framework for C#/.NET [2][8] over LAN and the Internet are presented. The framework is modeled using object oriented analysis and design techniques, and also uses object oriented implementations. Object orientation at implementation level provides interesting solutions such as automatic-proxy switching for object mobility, implementation of multiple interfaces for services and factories for object creation.

In addition to the above services, support for object level mobility and retraction services are provided. Mobile objects introduced in the C# ARC framework are self-contained autonomous objects which can move from machine to machine, performing assigned task. While object move and perform tasks on remote machines, they remain addressable for the originator through connection, and also to the current context where the object moves. A migrated object may decide to hop to another machine or the originator application may retract the object, or the object may be pushed onto another machine in its next hop by the originator or by its current context.

ARC framework over the Internet is an extension to ARC framework over the LAN. ARC over the Internet reuses the software components from the implementation of ARC over the LAN. Abstraction given to the developer of applications over ARC framework is same in both LAN and the Internet versions of ARC. ARC over the Internet uses ASP.NET XML webservices infrastructure for communication between machines on the Internet.

1.1 Scope of Work

First design of ARC implementation on heterogeneous cluster of workstations is discussed in [11]. In [10], Aruna et al. designed ARC using RPC based communication. Design was made object-centric at interface level but the actual implementation was procedural. They addressed distributed concepts like dynamic task distribution, fault tolerance, scalability and heterogeneity.

Design work and implementation has been redone on .NET/C#. Mobility of objects is introduced into ARC and supported at the programming language level. ARC framework is extended to support object hopping. It also enables passing messages to the migrated object from originator of the object and from remote environments.

ARC framework is extended to work over the Internet, nodes on the Internet can dynamically join and leave the ARC system. Distributed programs developer may write applications using ARC constructs to run over the Internet.

1.2 Organization of the Report

This report presents the design and implementation of the services provided by ARC for both LAN and the Internet versions. This report also presents example applications in detail for the applications developer over the ARC. Chapter 2 presents ARC framework features and ARC software architecture for the LAN environment. Chapter 3 discusses application development issues, it also describes user level of ARC framework. Design and implementation of ARC kernel over the LAN are presented in Chapter 4. Issues involved and a solution architecture to extend ARC over the Internet are discussed in Chapter 5. Design and implementation of ARC kernel over the Internet are presented in Chapter 6. Example applications are given in Chapter 7. Conclusion and guidelines for future work are given in Chapter 8. In Appendix A description of steps in installing ARC software is given and Appendix B describes steps in writing a *Hello World* application over ARC framework.

Chapter 2

ARC Framework Architecture

ARC framework capabilities and its architecture are presented. Section 2.1 presents framework capabilities. Section 2.2 shows architectural view of ARC and Section 2.3 presents a deployment view of ARC over LAN environment.

2.1 Framework Capabilities

Frameworks and architectures such as PVM [1], CORBA [6], COM/DCOM [9] have introduced interesting mechanisms for performing computations collaboratively over clusters of workstations. Some examples of these mechanisms are typed interprocess communication, synchronization mechanisms, remoting, naming services and remote method invocations. Frameworks for mobility such as *Aglets* [7] and *Voyager* [12] implement mobility at object level. As compared to various available message frameworks in these categories of parallel, distributed and mobile computing frameworks, the ARC framework on .NET discussed in this report specializes from the point of view of integrating anonymity, service orientation, mobility and remoting in a distributed scenario. Lightly loaded participant machines may join an ARC system dynamically or leave dynamically. An ARC computation may make use of these services to benefit from available computing capacity in a network. A horse power factor service and a fault tolerance service are provided as support services for computing in presence of load and failures. The framework is also extended to support multiple hopping of objects (including state and code) and remoting abilities to ARC objects.

Figure 2.1 shows a use-case diagram bringing out the functional view of the ARC framework. Two kinds of actors namely *Parallel/Distributed Application* and *Node Administrator* can be noted. Parallel or distributed applications use constructs provided by ARC system to develop programs involving more than one machine. Node administrator deals with join and leave services. The functionalities identified in the use case diagram are elaborated below.

- Specifying an ARC object: An object is specified as an ARC object through inheritance. An ARC object obtains all the capabilities provided by the ARC system.

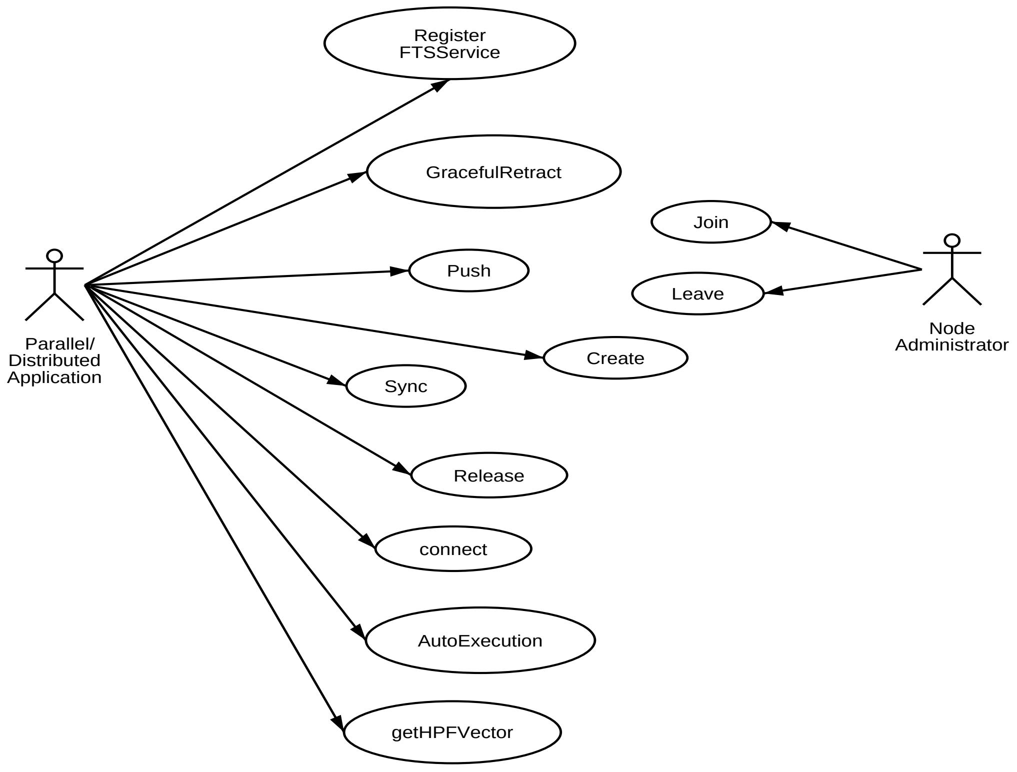


Figure 2.1: Use-Case Diagram for ARC

- Anonymity in Node selection: An application may select a machine by its *Horse Power Factor (HPF)* value, while the machine remains anonymous for the application. HPFs may be obtained via a call to *getHPFVector()* to a local HPF server. An HPF value is an instance of a class. An HPF server communicates with other HPF servers in the network.

- Explicit node selection: An application may also select a machine explicitly for migration through a call to *getHPFValue() on local HPF server.*

- Migration Assurance: When an application obtains an HPF value of a machine, or a vector corresponding to a set of machines, each value represents an assured slot for receiving a migrating object. The slots are unlocked through a call to *release()* on the local HPF server.

- Migration: Migration of an ARC object to an anonymous machine. An ARC object may be migrated by calling a method *push()* on the object. The method requires an HPF value corresponding to an anonymous machine as its parameter.

- Trigger: A method body to be executed after the object moves to remote machine is implemented as an overridden implementation of method *trigger()* by an ARC object.

- Parallelism: While an application migrates an ARC-object to an anonymous remote machine due to a call to *push*, it may continue in a non-blocking fashion. A result of remote execution available as migrated object's state.

- Auto-retraction: Asynchronous return of an object after completing the task at remote machine is a default retraction mechanism. After the object retracts, all invocations on the object are performed locally. The application remains unaware of the location of the object due to an internal automatic proxy switching mechanism.

- Wait till retraction: An application may wait on an ARC object till it is retracted through a call to *sync()* on the ARC object. If the object is already retracted, the call is unblocked immediately.

- Explicit Retraction: An object may be called back to originator when required through a call to *GracefulRetract()* on the ARC object. The remote ARC object is intimated via a flag which may be checked through a method *isRetractionSet()*. The trigger method in an ARC object may be implemented to handle an incoming graceful retract request.

- Roaming: An object may re-hop over a network of machines by means of a call to *push()* on the object. This call can be made by the originator, by its current context or by the object itself. The protocol followed by *push* is the same as that followed on its originating machine. For the originating application code, the object's location need not be known for call invocations on the roaming object. To establish this communication, the originating application needs to call a *connect()* invocation on its local ARC object handle. Proxy switching is performed internally in response to connect.

- Fault Tolerance: A desired fault tolerant behavior for an ARC object may be specified. Before an ARC object is migrated, to access a failure recovery service, the object is registered with a fault tolerance service (FTS) through a call to *register()* specifying the desired fault tolerance semantics for a given ARC object. An FTS server communicates with remote FTS servers for failure detection. Upon a failure detection, the FTS carries out resend operations through local ARC system interface.

- Communicating Mobile Objects: Roaming objects may exchange messages between each other. If the location of an object is known, an explicit proxy to the object may be obtained. For communication among roaming objects their locations have to be known to each other. An explicit proxy is required for all contexts other than the originating code for communication with a roaming object.

- Deactivation: An ARC object may be serialized to a harddisk file for later use. Applications may deserialize the object from the file to work with it.

- Reactivation: ARC object may be reactivated from serialized state on harddisk. Reactivation and Deactivation together enables working in disconnected mode of operation.
- Dynamic join and leave: Machines may join or leave an ARC system dynamically through a Join service and a Leave program. Every active participant exports a copy of the Join server. The list of currently active nodes is updated by join service on every machine.

2.2 Architecture

Figure 2.2 depicts the architecture of ARC framework. The ARC system is organized in the four layers which are described below.

2.2.1 ARC User Upper Layer

This layer consists of user programs running on the ARC system. These programs are distributed applications or node administrators. Distributed applications use services meant for user applications. Node administrators are responsible for joining a machine into the ARC network as well as disconnecting the machine from ARC network.

2.2.2 ARC User Lower Layer

Classes in this level include proxies for the ARC services and classes which may be generated through a preprocessor operating on interface descriptions. The inheritance structure of these classes is shown subsequently. Class *Real* is at the lowest level of inheritance and user implements the interface in this class. This layer also consists of proxies to services provided in ARC kernel level. These are obtained using .NET remoting infrastructure.

2.2.3 ARC Kernel Layer

This layer provides services to the layers above it. Depending on the user of the services, these services are classified into 3 sub parts.

- **ARC Object Services:** It allows registration of newly created ARC objects, Migration (send and receive) of code and state, and activation and execution of trigger on a newly arrived remote ARC object. When a migrated ARC object returns asynchronously, the object is inserted into its original location by proxy switching through ARC object proxy layer. ARC object proxy layer consists of proxies to locally registered ARC objects. The proxy layer is essential since the ARC object services are located in a different address space than that of the user programs.
- **Developer Services:** Distributed or parallel programs uses these services through proxies, which can be obtained using .NET remoting framework. These services can also be used from ARC user lower layer. Services supported in this category are

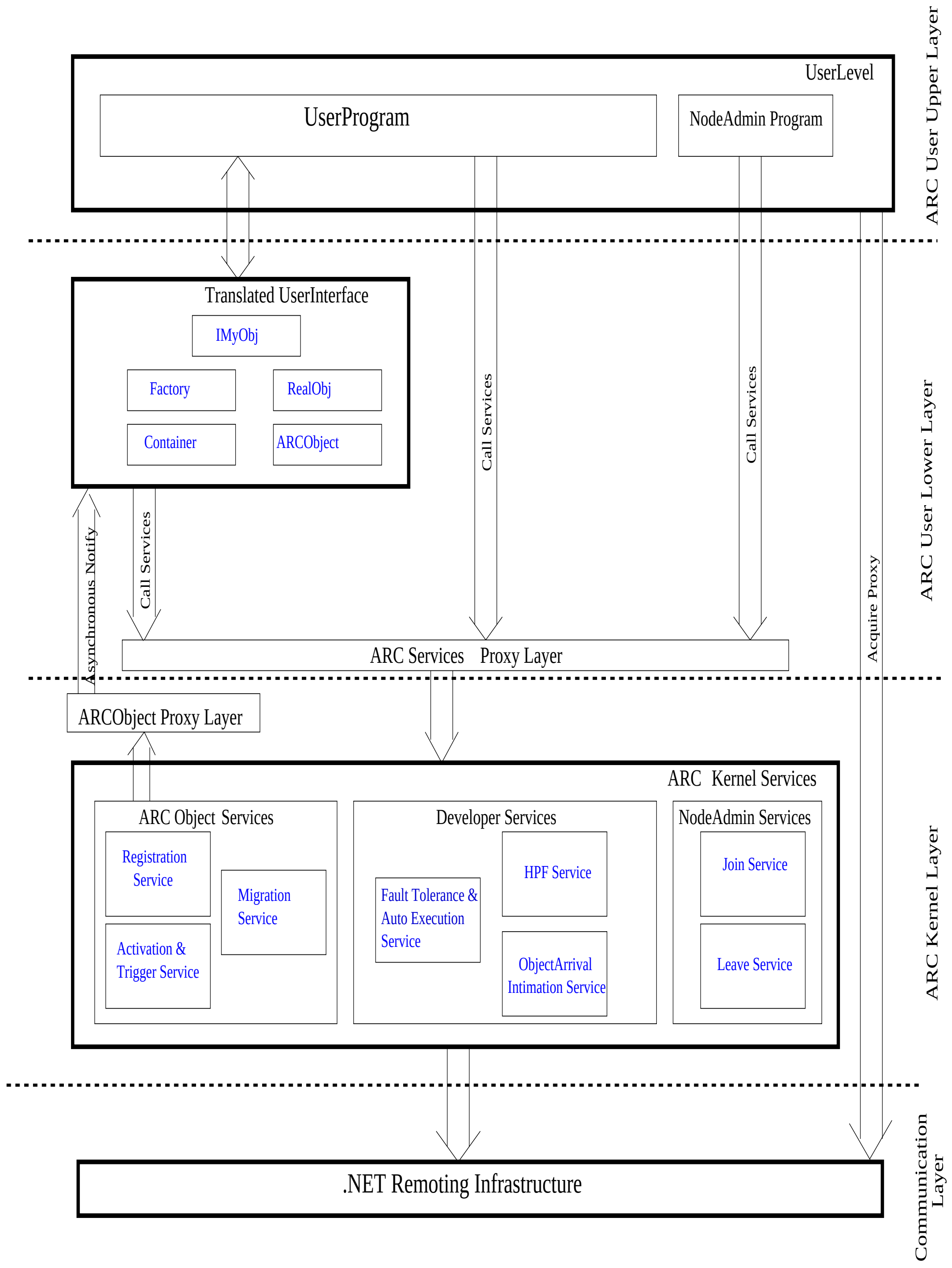


Figure 2.2: ARC Framework Architecture

fault tolerance and auto execution, Horse Power Factor and remote slot locking and unlocking, and asynchronous object arrival intimation services.

- **Node Administrator Services:** Node administrator can join a machine into the ARC network and also can unsubscribe a machine from the ARC network. Node administrator services include join and leave services. An active ARC computation uses actively participating nodes at a given time.

2.2.4 Communication Layer

.NET provides remoting infrastructure which allows method invocation on a remote machine. Communication between any two machines takes place through this level. The .NET services accessed in this layer consists of registration of services and creation of proxies.

2.3 ARC Deployment View

Figure 2.3 shows deployment view of ARC software over the LAN. A node in the LAN may join ARC system to give its computational resources to other nodes present in ARC network. Joined nodes form a logical ring to handle fail stop failures in ARC network. Nodes in the ARC network contain ARC software components and provide different ARC services accessible to remote machines through .NET remoting infrastructure over TCP. Chapter 4 discusses the details of ARC software components and its services.

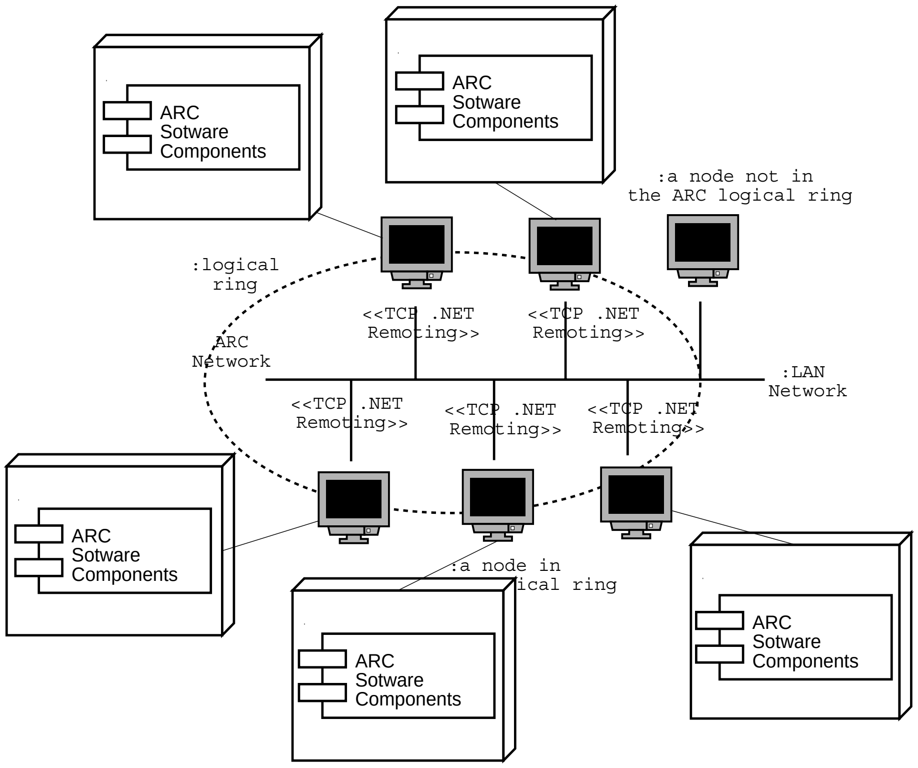


Figure 2.3: Deployment View of ARC Software

Chapter 3

Application Development using ARC

Application development process using ARC, and design and implementation of higher level in the ARC framework are discussed in detail. Section 3.1 presents application development process. Section 3.2 discusses the design and implementation techniques of ARC user level.

3.1 Application Development Process

An ARC application may be classified into broad categories of distributed, parallel and mobility based applications. An application may also combine one or more of these features. Applications may involve autonomous objects migrating over the network and exchanging messages. These distributed objects carry unique identities and may work co-operatively to serve a common purpose.

In an application involving parallelism, a large computation may be divided into small individual subcomputations expressed as ARC objects. Using ARC constructs, these subcomputations may be executed on remote machines in parallel. Results of remote executions arrived at the originator asynchronously as ARC objects retract. Earlier work on parallelism on an ARC platform is discussed in [11]. An application may also employ multiple hopping of an ARC object to complete its work-flow. ARC system allows ARC objects to be migrated over a specific path.

This section highlights a development process of ARC programs through an example. In the example discussed, an ARC object is migrated to a remote anonymous machine, where a specified task is triggered on arrival. The object retracts asynchronously to originator context after completion of the triggered task.

Interface Specification

The development process begins with a specification of an ARC object interface as shown below. The interface consists of public member functions that the ARC object exports.

These member functions are exported in addition to default ARC object members such as a trigger executor. In the interface specification below, the ARC object interface inherits a library interface *IRefCount* to add reference counting feature to ARC objects of type *IMyObject*.

```
public interface IMyObject : IRefCount { // User specified interface
  void task(); // a public member of the ARC object
}
```

Implementation of ARC Object

Implementation of the interface is done in class *Real* in the namespace corresponding to user specified ARC interface. Skeleton of this class is generated from the ARC object interface. The architecture of the generated classes is discussed in next section.

```
namespace NSIMyObject{
   [Serializable]
   public class Real: PReal{
      public override void Trigger(){
         Console.WriteLine(‘‘This Message is Expected
                              to be Displayed at Remote Node’’);
         this.task(); //method to print Hello World!
      }
      public override void OnReturn(){ }
      public override void OnRetract(){ }
      public override void task(){
         Console.WriteLine(‘‘Hello World!’’);
      }
   }
}
```

During implementation of *Real*, definitions of member functions need to be stuffed in by the programmer. Method *Trigger*() is automatically executed after the object migrates. In this example, method *task()* is called from within *Trigger()*. The code for class *Real*, which implements interface *IMyObject* is shown below. Methods *Trigger()*, *OnReturn()* and *OnRetract()* are due to inheritance from interface *ITrigger*. The methods in interfaces *ITrigger* and *IMyObject* are to be implemented in class *Real*.

Originator Application

Below is an example originator code. The originator program invokes a creation request on the factory class available under the generated namespace corresponding to the ARC object interface. After creation, the program sends instance of class *Real* to an anonymous remote machine and executes a local method in parallel. Finally the program blocks through a call to method *Sync()* till the migrated ARC object retracts.

```
namespace testHello{
  public class HelloClass{
    public static void Main(){
    // 1. Connect to local HPFServer
       UserInterface_HPFVector.IUser hpfvector;
       UserInterface_HPFServer.IUser hpfserver =
            (UserInterface_HPFServer.IUser)Activator.GetObject(
                        typeof(UserInterface_HPFServer.IUser),
                        "tcp://localhost:8105/HPFServerClass" );
   // 2. get HPFVector
       hpfvector = hpfserver.getHPFVector(1);
       int i  = hpfvector.SizeOfHPFVector();
       UserInterface_HPFValue.IUser hpf1=null;
       hpf1 = hpfvector.getHPFValue(0);
   // 3. Instantiate an ARCObject
       NSIMyObject.IContainer arcobject = NSIMyObject.Factory.New();
   // 4. send created object to remote machine
       arcobject.push(hpf1);
   // 5. do any work in parallel
       Console.WriteLine(‘‘to be executed parallelly’’);
   // 6. wait for object to come back
       arcobject.sync();
   // 7. end of program
       Console.WriteLine(‘‘the end’’);
    }
  }
}
```

Note that the HPF value obtained through the HPF service represent a machine on which the ARC object migrates. The remote machine remains anonymous to the originator application program. After executing a predefined task in its trigger specification the object retracts. In the meanwhile the originator performs an activity in parallel.

3.2 Design of ARC User Lower Level

Figures 3.1 and 3.2 show a view of the hierarchy generated from the given ARC object interface. The classes and interfaces are classified into two categories of generated classes and interfaces, and library classes and interfaces. The user application programs use some of the generated classes either *as it is* or by *specialization and stuffing*. A naming convention is used to name the generated classes. Instead of naming each of the classes on user specified ARC object's interface, they are named with a standard scheme, but are placed in a namespace. The namespace is in turn named on the ARC object's interface. This

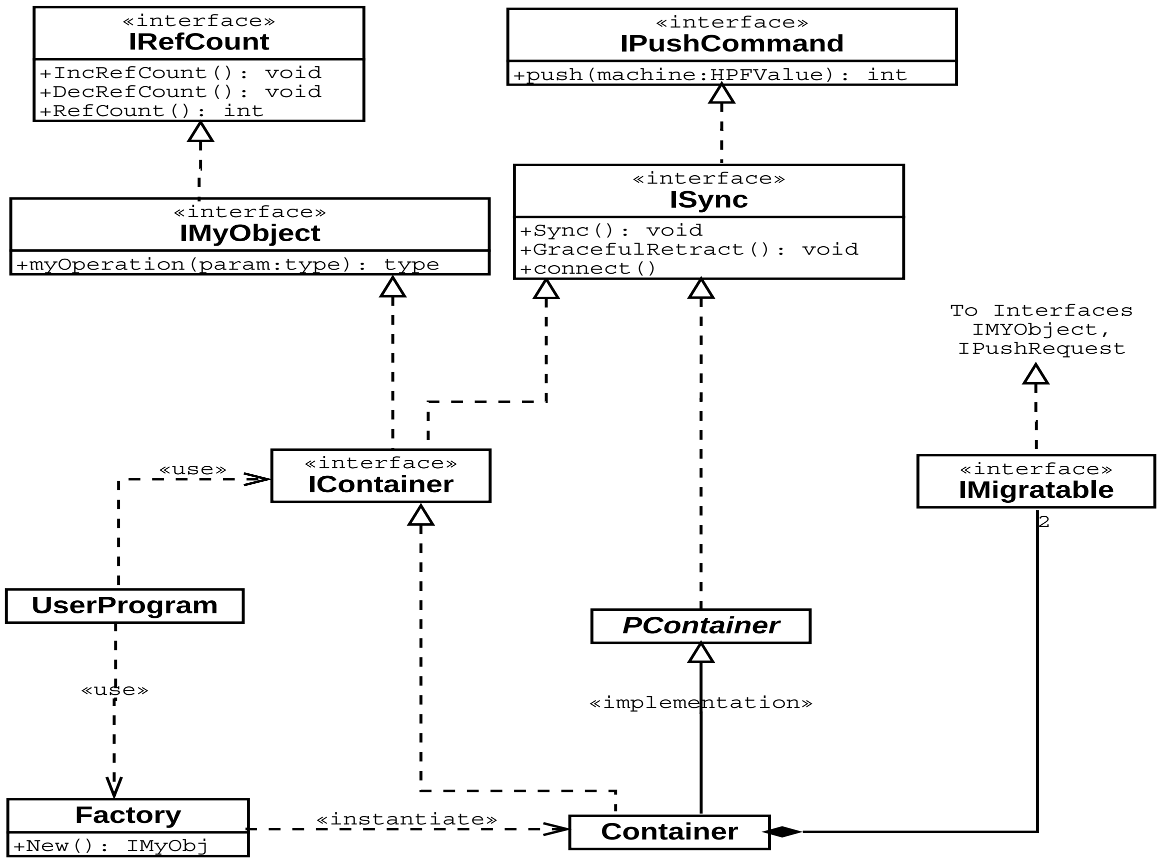


Figure 3.1: Container

convention is employed for the convenience of application programmers in handling various interfaces and classes in an ARC program. The figures omit the details of namespaces. The generated classes also reuse interfaces and implementations provided in the library. The design of the hierarchies is discussed below.

3.2.1 Interfaces and Generated Classes

In this section, we will discuss the design of the generated classes, generated interfaces and library interfaces, since these are concerned with application programs.

- Interface IMyObject: This interface represents the original interface description specified by the user. The user may access an ARC object through this interface polymorphically. In such a case, only the member functions on the interface are visible (e.g. member function *task*(), and reference counting in this case). This interface is to be implemented by the applications programmer.

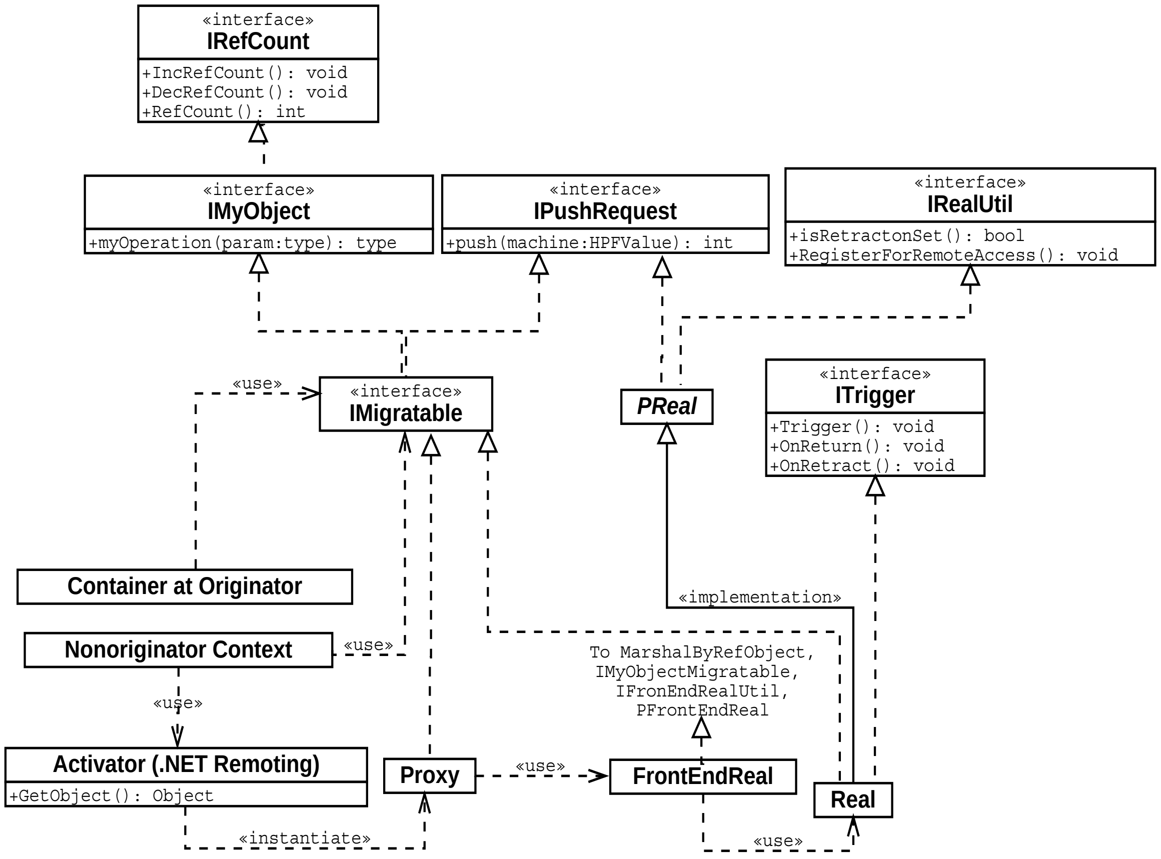


Figure 3.2: Proxy and Real

- Interface ITrigger: For every ARC object, this interface needs to be implemented by the applications programmer. The interface includes method *Trigger()*, which is called upon migration to a remote destination. Similarly, methods and *BeforeRetract()* and *OnReturn()* are respectively called before the object retracts from the remote machine and after it returns to the originator.

- Interface IRefCount: It is useful in keeping track of number of external actors that are referencing an ARC object. Implementation of members of this interface may be provided in class *Real*. It is optional in the inheritance hierarchy and including it in the hierarchy is left to the programmer.

- Class Real: This class is to be implemented by the applications programmer. This class shown in Figure 3.2 implements interfaces IMyObject and ITrigger. Observe that figures 3.1 and 3.2 shows interface IMyObject as a subclass of interface IRefCount. Reference counting has been a common feature used in component programming on

COM/DCOM models [9] to keep track of usage. This feature is useful for garbage collection, especially to take care of references that an object may accumulate while hopping from machine to machine. A library implementation of reference counting may be used in class *Real* through inheritance.

- Class Container: User program gets a handle to an instance of this class by invoking method *New()* on class *Factory*. Class container acts as a handler to the real ARC object which may migrate. It inherits from interface *IMyObject* through interface *IContainer* to enforce the abstraction of the user specified ARC object. The container either holds the real object when it's resident, or holds a proxy to it when the object migrates. It delegates the function invocations made by the user to the real instance of class *Real* which may be available locally or may be migrated to a remote machine. It employs an automatic proxy switching mechanism which makes makes migration transparent to a local application program. After the migration, the application program may invoke a method *connect*() on the container, and subsequently can invoke member functions on it as if the ARC object is available locally. The container reuses a library implementation *PContainer* through inheritance as shown in Figure 3.1.

- Class Proxy: The class as shown in Figure 3.2, represents proxy to an instance of class *FrontEndReal*. It's instances are obtained through .NET remoting infrastructure on invocation of method *Activator.GetObject()*. A proxy instance may be obtained from any context on any machine provided that a request is made to activator where the ARC object registers its front end for allowing remote accesses to it.

- Class FrontEndReal: An instance of this class wraps the real ARC object when it migrates to a machine. A front end is required to make a migrated ARC object accessible to other contexts for public method invocation. It can be noted that in order to make an ARC object migratable, class Real object does not inherit from MarshalByRefObject, which is required for making it remotable. Hence, to allow remote access to a migrated ARC object, a front end is used as a mechanism to make a migrating object remotable. This class delegates user requests to *Real*. When method *RegisterForRemoteAccess()* is called from inside *Trigger()*, an instance of class *FrontEndReal* is created internally, and is initialized to point to the calling real object, thus publish itself as a remotable object. As shown in Figure 3.2, class *FrontEndReal* is inherited from the library class *PFrontEndReal* which provides its implementation partially.

- Class Factory: Application programs involving an ARC object use this class for creating instances of the ARC object. Application programs uses interface *IContainer* to hold the instance returned by method call *New()* on class *Factory*. Factory returns an instance of class *Container*.

3.2.2 Library Classes

The classes in this category provide partial or full implementations for some of the interfaces for classes Container, Real and FrontEndReal. These library classes include classes *PContainer*, *PReal* and *PFrontEndReal*.

- Class PContainer: As shown in Figure 3.1, this class realizes interface *ISync*. Member functions on the ARC object for the use of the originator application are implemented in this class. They include member functions *push()* from interface *IPushCommand* and member functions *Sync()*, *GracefulRetract()* and *connect()* from interface *ISync*. Interface *ISync* and *IPushCommand* are visible to the originator application through interface *IContainer*, as it can be traced through the class diagram in Figure 3.1.

  Method *push* is used for migration of an ARC object. Method *sync* blocks the caller till the object returns to the originating application. Method *GracefulRetract* sets a flag on a migrated ARC object which may be checked by ARC object's executing code (such as *Trigger*). ARC object may return from trigger on observing this flag to be *true*. Originator application uses method *connect* to establish connection to migrated ARC object if the latter is registered for remoting after migration. After a successful connect, application code may continue to invoke public member functions on the migrated ARC object through its container.

- Class PReal: As shown in Figure 3.2, this class realizes interfaces *IPushRequest* and *IRealUtil*. Class *PReal* implements methods that can be invoked from within the executing code of a migrated ARC object, such as through *Trigger*. These methods include *push()* from interface *IPushRequest*, *isRetractionSet()*, *RegisterForRemoteAccess* and *UnregisterForRemoteAccess* from interface *IRealUtil*.

  Method *push* from interface *IPushRequest* allows a migrated object to re-hop to another remote machine. However, this non-blocking request is fulfilled only after completion of an executing trigger. Method *isRetractionSet* is used to check the status of retraction flag which is settable from originating application code as discussed above. In addition to these two methods, a migrated ARC object may register itself for remoting through *RegisterForRemoting* immediately after migration, and unregister just before migration or retraction. These member functions are available for an ARC object, and their use is optional.

- Class PFrontEndReal: As shown in Figure 3.2, this class inherits interface *IPushRequest*. The implementation is used in class *FrontEndReal* to delegate calls to method *push()* to the instance of class *Real*.

Chapter 4

Design and Implementation of ARC Kernel Level for LAN

In this chapter, ARC services, their design and implementation techniques are presented. The ARC framework on .NET has features of anonymity of participating nodes, service orientation, mobility of objects, handling fail stop failures and load measurement services. Participant machines may join an ARC system dynamically or leave dynamically. An ARC computation may make use of various services to benefit from available computing capacity in a network. A horse power factor service and a fault tolerance service are provided as support services for computing in presence of load and failures. The framework is also extended to support multiple hopping of objects (including state and code) and remoting abilities to ARC objects. Services of ARC can be categorized into three classes based on the user of those services. The three classes are, ARC object services, Developer Services and Node administrator services. Section 4.1 presents an implementation view of ARC software. Section 4.2 explains ARC object services, Section 4.3 presents developer services and Section 4.4 presents node administration services.

4.1 High Level Implementation View of ARC Software

Figure 4.1 shows UML component diagram [5] of ARC. It shows a node called *ARC Node*, which represents a machine/node in the ARC network. Each node in ARC network contains component instances shown in Figure. These components exchange messages among themselves to provide the supported services.

Table 4.1 categorizes components of ARC implementation into three classes depending on the type of service they provide. Description of these components is given below. The interface names of the components indicate their respective client components.

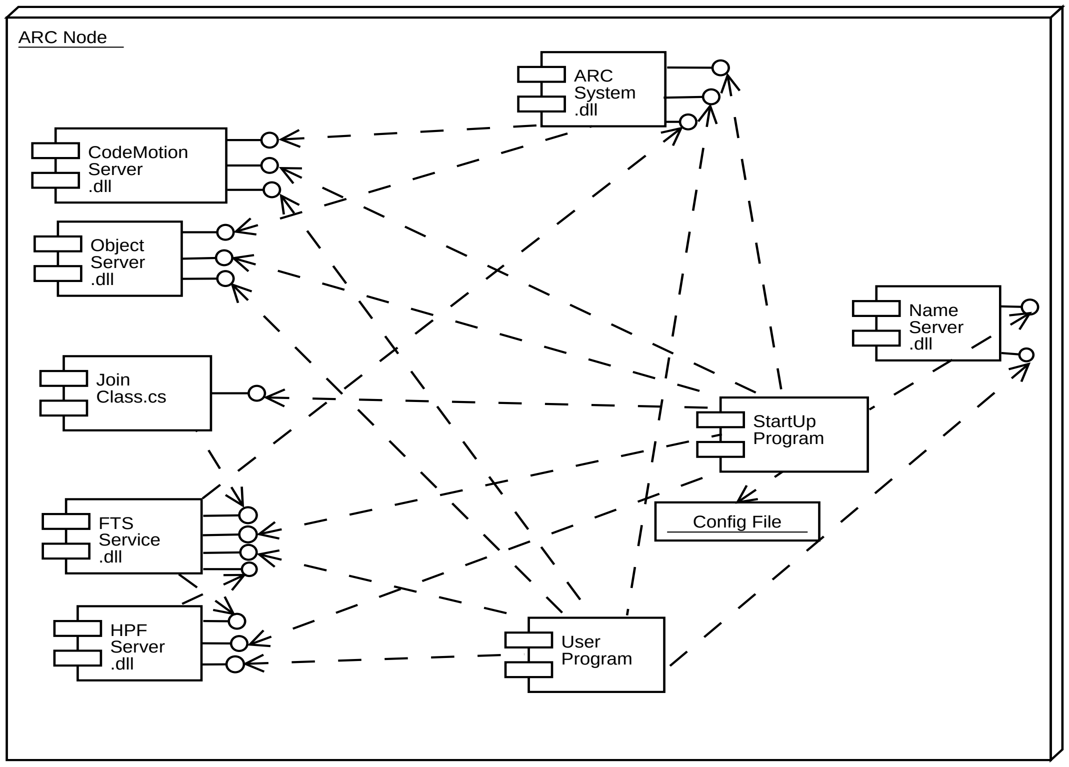


Figure 4.1: Component Diagram of ARC Software

ARCSystem

This component is used by UserProgram, FTSService and remote ARCSystem. It is responsible for maintaining registry of ARC objects. It redirects all the requests from local and remote environments to proper classes. *ARCSystem* starts new threads when an object comes for execution. It is also responsible for sending objects back to their originators after completion of task. The interfaces for this component are listed in Table 4.2.

ARC Object Service Components	Developer Service Components	Node Administrator Service Components
ARCSystem Object Server Code Motion Server	HPFServer FTSServer NameServer	JoinServer Startup Program

Table 4.1: Components Categorized based on Services

IARCObject
ObjID getObjID()
void Register(ObjID id, ARCSystemInterface_ARCObject.IARCSystem obj)
int push(Message msg)
void Retract(Message mesg)
void Hop(Message msg)
string WhichMachine(ObjID id, int pushno)
int WhichPort(ObjID id, string ip)
IFTSService
ArrayList getObjectsOnFailedNode(string ip)
void SendResendMessage(ResendMessage resendmsg)
void AutoExecuteLocally(ObjID objid)
void ReturnARCObjectToApp(ObjID objid)
bool completePendingWork( )
IPeer
int SendMessage(Message msg)
int returnPortNo(ObjID id)
ISerializer
int SendMessage(Message msg)
IThreadExecutor
void MessageforNextHop(Message mesg)
Startup functions
void BufferThreadStart()

Table 4.2: ARC System Class

Object Server

This component is used by *ARCSystem* and ARC object of *UserProgram*. This class is useful in implementing asynchronous return of migrated objects. It stores the objects that are returned from a remote node. *ARCSystem* invokes method *store_object* on this class by keeping an ARC object inside parameter. After receiving a notification from *ARCSystem*, *UserProgram* interacts with *ObjectServer* to load the returned object. The interfaces for this component are listed in Table 4.3.

NameServer

This component is used by ARC object and user applications on remote nodes. It is responsible for implementing object arrival intimation service. User programs can register their request for delivery of asynchronous intimations on incoming ARC objects. ARC object can register themselves with *NameServer* to allow remote context to start accessing it. The

IARCSystem
int store_object(Message msg)
IARCObject
ITrigger load_object(ObjID objid, int pushnumber)

Table 4.3: Object Server Class

intimation service is provided for remote machine and not for the originator machine. The interfaces for this component are listed in Table 4.4.

IARCObject
void Register(string classname, Object obj) void UnRegister(string classname)
IUserProgram
void RegisterRequest(string classname, Object obj)
Startup functions
void startThreads()

Table 4.4: NameServer Class

JoinServer

This class is responsible for allowing nodes to join the existing ARC system without having to interrupt the functioning of already joined nodes. The interfaces for this component are listed in Table 4.5.

IPeer
bool eachresultjoinrequest(JoinMessage reply) bool eachresultjoin() JoinMessage requestJoin(JoinMessage message) JoinMessage Join(JoinMessage message)
Startup Program
void boot()

Table 4.5: Join Server Class

Code Motion Server

This component is used by *ARCSystem* and ARC object. This component is responsible for storing code of an ARC object. *CodeMotionServer* in co-operation with *ARCSystem* implements code transfer. The interfaces for this component are listed in Table 4.6.

IARCSystem
void catchfile(ObjID id, Code code) Code getCode(ObjID id) void setLockValue(ObjID id,int lockvalue)
IARCObject
void LocalCopy(ObjID id, Code code)

Table 4.6: CodeMotion Server Class

Startup Program

This is responsible for starting all components of ARC kernel. *StartupProgram* uses configuration file to read configuration settings of the user, it uses this information while starting all the components. It starts various services one by one as per their dependencies and invokes the initialization member functions on them.

FTSServer

This component is used by *JoinServer* and *UserProgram.* This class is responsible for failure detection of one neighbor machine in a ring configuration (node id + ip address). It also maintains the information about all nodes in the ARC distributed system. *UserProgram* interacts with this class to register *ARCObject* for fault tolerance service. This class provides specified quality of fault tolerance as desired by the user. The interfaces for this component are listed in Table 4.7.

HPFServer

This component is used by *UserProgram* and *FTSServer.* This class is responsible for giving locks on local machine as well as acquiring locks from remote machines. This class calculates HPF value of the local machine. *UserProgram* as well as *FTSServer* are able to query this class for HPF values. HPF value is returned as an object of class *HPFValue*, which is a type <ip-address, lock-no, actual hpf value>. The interfaces for this component are listed in Table 4.8.

IUserProgram
int Register(Object obj , int howmany_resends)
IJoin
setOfNodes requestNodeSet_Join() void updateNodeSetWhenLeader(setOfNodes nodes) void updateNodeSetWhenJoin(setOfNodes nodes) void setNodeset(int myId, setOfNodes nodes)
ILeave
void LeaveBroadcast(LeaveMessage mesg) setOfNodes Leaving() int returnLocalnodeId()
IARCObject
int registerForAutoExecution(ObjID objid)
IPeer
int checkforliveness() int HandleFailure(FailureMessage mesg)
Startup functions
void start_thread()

Table 4.7: FTSServer Class

User Program

This is a distributed application, which makes use of constructs given by the ARC. It has a reference to the instance of ARC object, which is the application ARC object. *HPFServer*, *FTSServer* and *NameServer* are visible to the user applications. *ARCSystem* is accessed from inside ARC object and not visible to the high level user applications. *CodeMotionServer*, *JoinServer* and *ObjectServer* are helper components and not visible to the user applications.

4.2 ARC Object Services

ARC object services include registration of an ARC object, migration service and object hopping across network of machines. These include services to create an ARC object, send an ARC object from one machine to the other for multiple times across network of machines, activation and execution of trigger on a newly arrived remote ARC object. ARC provides constructs to wait for an object to return to originator machine and construct to retract an object when required from originator application. ARC also provides construct to connect to the migrated object from originator application and allows to exchange messages with it.

IUserProgram
UserInterface_HPFVector.IUser getHPFVector(int size)
UserInterface_HPFValue.IUser getHPFValue(string machaddr)
UserInterface_HPFValue.IUser atMostOneHPFValue(string[] machaddrarr)
int Release(Object hpfvalue)
IARCSystem
Object GetHPFValue(string machaddr)
ARCSystemInterface_HPFVector.IARCSystem GetHPFVector(int size)
IARCSystem, IFTService
Object GetHPFValue()
IPeer
HPFServerInterface_HPFValue acquireLock()
ILeave
void doNotgiveLocks()
Startup functions
void start_thread()

Table 4.8: HPFServer Class

4.2.1 Registration Service

A task is a unit of computation which can be executed at any machine. In ARC, a task is modeled as method *Trigger* inside ARC object. Figure 4.2 shows the sequence diagram for creating a task. *UserProgram* requests *Factory* to create ARC object. Class *Factory* creates an instance of class *Container*. Constructor of class *Container* submits its code to local *CodeMotionServer* and registers itself with *ARCSystem*. *ARCSystem* assigns a unique identifier to each ARC object.

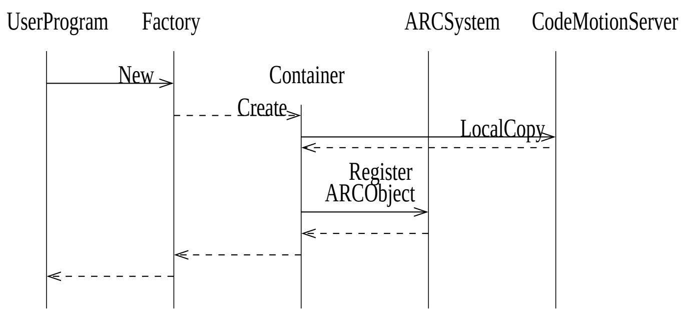


Figure 4.2: Sequence Diagram for Task Creation

4.2.2 Migration Service

ARC supports parallelism at the programming language level. ARC provides two constructs namely *push* and *sync* to send a subtask to remote machine and to wait for the return of the migrated object. A subtask is an ARC object with a trigger method implemented by the user.

Push Operation

This section describes working of *push* operation. Figure 4.3 shows UML sequence diagram illustrating the working of *push* method invocation.

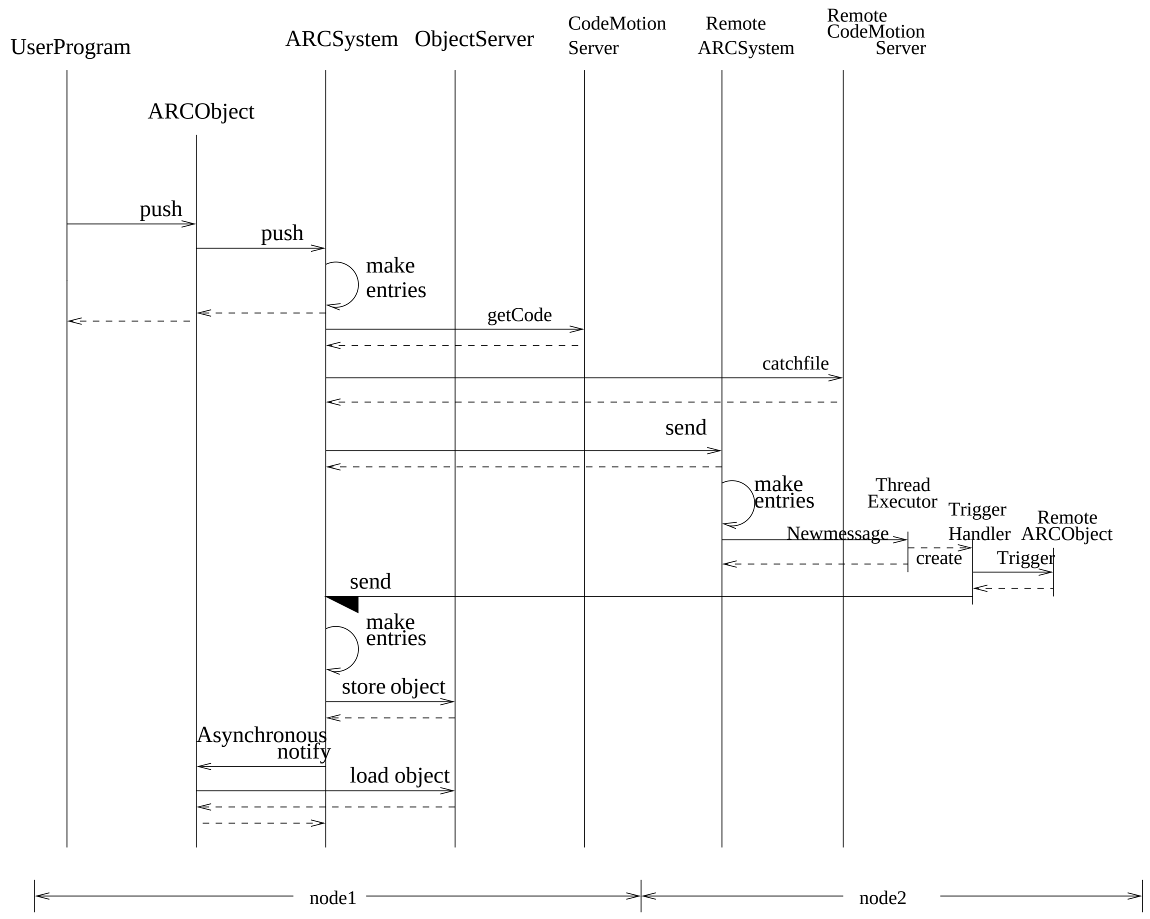


Figure 4.3: Sequence Diagram for Nonblocking Push Operation

Push is a non-blocking call such that user can proceed with other computation while *Trigger* method executes on remote machine. User has to submit a lock present in *HPFValue* returned by a call to *getHPFVector*, while calling *push* method as a parameter to the method.

Procedure to acquire a lock is described in Section 4.3.1. Local *ARCSystem* makes proper entries in its data structures such as machine address to which an ARC object is about to be sent. *ARCSystem* packs the object in a message and sends it to the remote machine.

Asynchronous Return

Migrated object can asynchronously come back after completing execution of method *Trigger* at remote node. *ARCSystem* receives asynchronous arrival of the object, stores the object in *ObjectServer* and notifies the local ARC object container using registration information. Subsequently, ARC object calls method *load_object* on *ObjectServer* to retrieve the object.

Synchronization

Synchronization is one of the primitive operation that must be addressed by any distributed programming language.

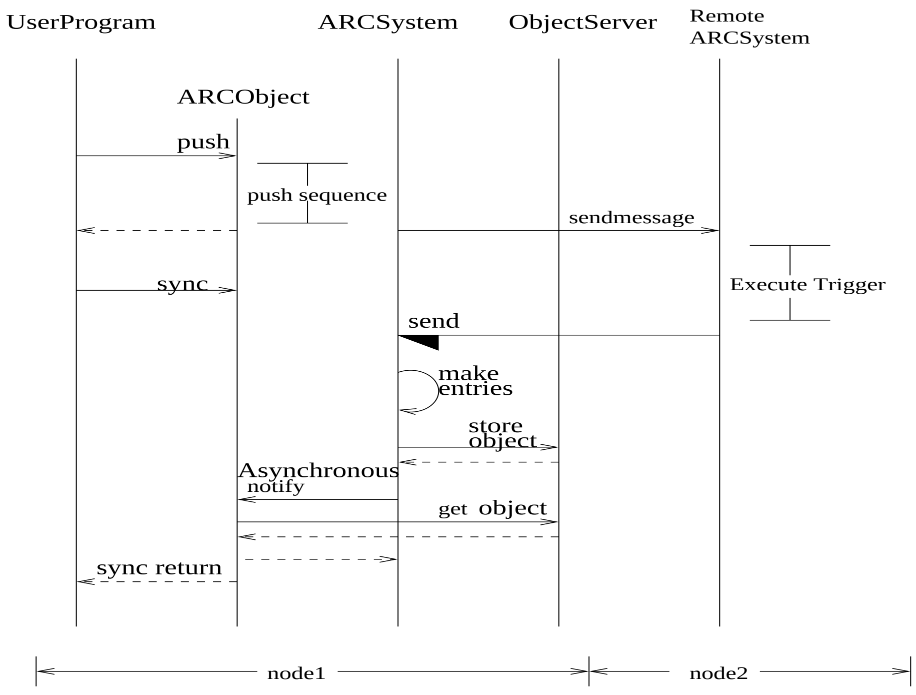


Figure 4.4: Sequence Diagram when Sync is Called Before Return of Migrated Object

In the sequence diagram shown in Figure 4.4, method *Sync* is called before return of migrated object and this call will be blocked until object comes back. If *sync* is called after return of migrated object then that call will be returned immediately. In any case, it is assured that state of the object is synchronized to current state and object is local when *sync* call returns.

4.2.3 Object Hopping

Other facet of ARC system is that it can act as a framework to support object mobility. Using *GracefulRetract* construct originator of ARC object can bring back the migrated object whenever required. ARC permits the originating application to execute methods on migrated object at remote machine by connecting directly to the object. ARC also supports multiple hopping of an object. An object can hop through multiple machines performing some assigned task.

Retraction Construct

GracefulRetract operation on an ARC object brings back the migrated object from remote machine. User may implement *OnRetract* method in class *Real* and this will be called at remote machine by remote *ARCSystem* before sending the ARC object back. User may use a function *isRetractionSet* to programatically check on remote machine inside trigger method, whether retraction is requested or not. It is users responsibility to check for retraction inside *Trigger* method. User has to program the *Trigger* method in such a way that if retraction is set, execution of *Trigger* method should terminate. After execution of *Trigger* terminates remote machine invokes method *OnRetract* on ARC object and sends it back to the originator.

Connect Construct

Connect construct is used to connect to the remote ARC object. After calling *push* method, user has to call *connect* method to be able to exchange messages with the migrated object from the sending machine.

Figure 4.5 shows the interaction diagram illustrating the working of connect construct. *UserProgram* calls push on the ARC object to send it to a remote machine. ARC object registers a proxy to itself by calling method *RegisterForRemoteAccess* from *Trigger* method. Originator program may call method *connect* on the ARC object to get connected to the object, wherever it is present. When method *connect* is called on local ARC object, it requests .NET Remoting framework to get a proxy reference to remote proxy. ARC object subsequently calls method *setProxyConnected* on itself to direct any further messages on it from local *UserProgram* to remote proxy. This is called **automatic proxy switching** since all incoming messages to ARC object at originator machine are directed to remote machine using acquired proxy. *UserProgram* sends messages to ARC object in the same way before and after calling *connect* construct but the messages are processed locally in one case and remotely in other case respectively. If method *connect* is called before ARC object registers a proxy from inside *Trigger* method at remote machine raises an exception.

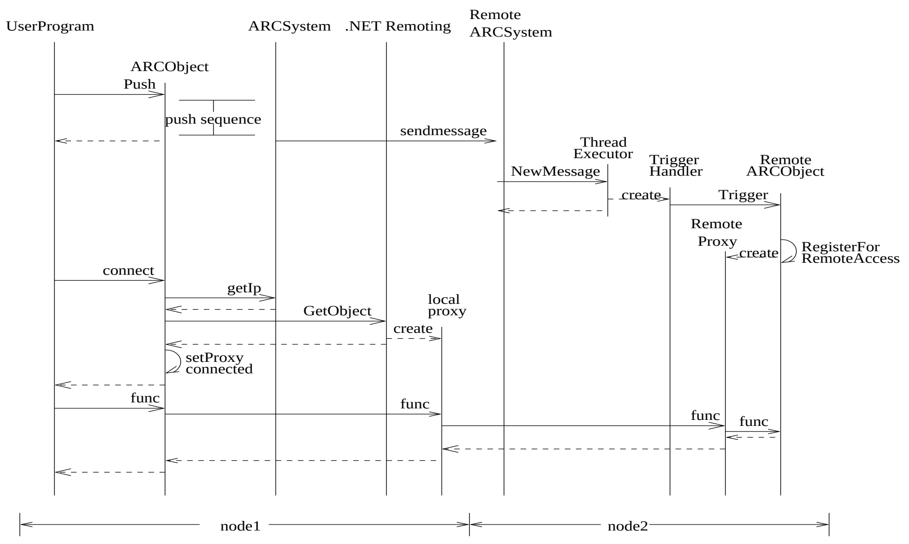


Figure 4.5: Sequence Diagram Showing Working of Connect Construct

Multi-Hopping

ARC object can migrate from one machine to another over the network of machines before returning to the originator. While roaming, it can carry current state of the object and resume the operations at the new machine. This is referred to as multiple hopping of an object.

Once ARC object has been pushed to some remote machine, it performs computations specified in *Trigger* method. Following three actors has the ability to push a migrated ARC object on to next machine.

1. Originator of the ARC object.
2. Current environment holding the object.
3. Object itself.

First two actors mentioned above use proxy to execute *push* method on the ARC object. Third actor (i.e. ARC object itself) calls *push* method from inside *Trigger* method.

The feature of multiple hopping is to treat an ARC object as autonomous migrating object. ARC object should not change its location while any application holds a reference of its proxy. Any application that wants to use ARC object should increment reference count of the ARC object by one and after finishing using the object it should decrement the reference count by one, so that no other program can change the location of the ARC object while the application is using the object through proxy.

4.3 Developer Services

ARC provides various services for distributed application program developers. ARC developer services include Horse Power Factory (HPF) service, Fault Tolerance Service (FTS), Object Arrival Intimation service and, Activation and Deactivation of an ARC object. This section presents design and implementation these services.

4.3.1 HPF Service

One of the main reasons of using distributed processing is to achieve speedup. Parallelism in a distributed environment may be achieved by executing different independent tasks on different machines simultaneously. ARC addresses parallelism at programming language level by giving non-blocking object migration constructs. One of the key aspects in writing parallel programs on a loaded cluster is dynamic load adaptability. Dynamic load adaptability is the ability to select a lightly loaded node at runtime for executing a parallel subtask. The HPF service provides a metric to measure the effective processing ability of a machine at a given time.

HPF Service Protocol

This section describes design of HPF service protocol using UML class diagram. Figure 4.6 shows the class diagram for *HPFService*. *HPFServer* class contains *LocalData*, it is responsible for maintaining locks and calculating HPF value of the local node.

In response to *getHPFVector* call made by *UserProgram* using *IUser* interface, *HPFServer* makes *acquireLock* request on remote machines on behalf of user and returns the received *HPFValue* containing lock in the form of a vector. Structure *HPFValue* is a type <ip-address, lock-no, actual hpf value> and *HPFVector* is an array of *HPFValue*s. User is allowed to look into HPF value of individual items in the vector. *HPFServer* uses *IPeer* interface to call *acquireLock* method on remote peer *HPFServers*.

Besides the user program, another user of HPF service is *FTSService*. *IFTSServer* interface on *HPFServer* is used by *FTSService*. If a remote machine fails after user sends a task to it then *FTSService* is responsible for resending the same task to another available machine. *FTSService* requests local *HPFServer* using interface *IFTSService* shown in Figure 4.6 to get a lock on any available machine.

Processor Selection

Processor selection deals with the ability to select a node to execute a parallel subtask. Processor selection can be done statically and/or dynamically. ARC supports both static and dynamic node selection. Static node selection can be done by hand-coding IP address in the application program. Dynamic node selection is done as follows.

Dynamic node selection is the ability to select a node at the runtime of user application. ARC gives a programming construct called *getHPFVector* to acquire a lock on remote

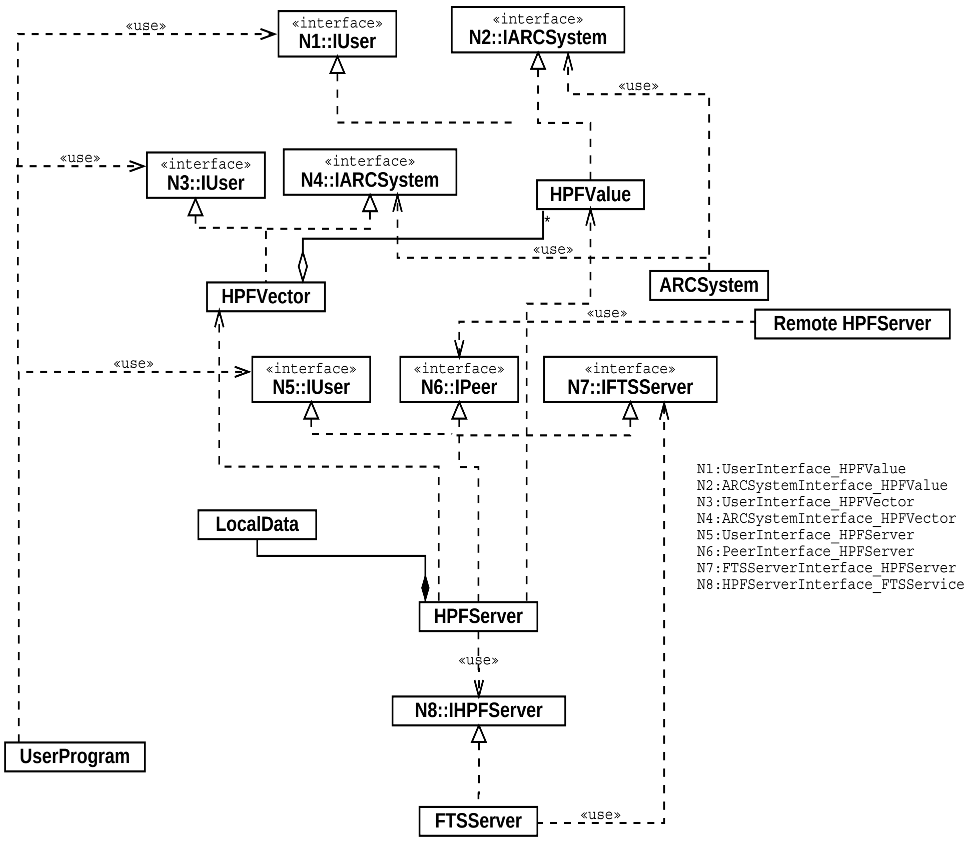

«use»
«interface» N1::IUser
«interface» N2::IARCSystem
«use»
«use»
«interface» N3::IUser
«interface» N4::IARCSystem
HPFValue
*
«use»
HPFVector
ARCSystem
«use»
Remote HPFServer
«use»
«interface» N5::IUser
«interface» N6::IPeer
«interface» N7::IFTSServer
N1:UserInterface_HPFValue
N2:ARCSystemInterface_HPFValue
N3:UserInterface_HPFVector
N4:ARCSystemInterface_HPFVector
N5:UserInterface_HPFServer
N6:PeerInterface_HPFServer
N7:FTSServerInterface_HPFServer
N8:HPFServerInterface_FTSService
LocalData
HPFServer
«use»
N8::IHPFServer
UserProgram
FTSServer
«use»

Figure 4.6: Class Diagram for HPF Service

machine. A lock on remote machine is used by a user program to send a task onto that machine. Acquired lock also contains HPF value [10] of the machine. HPF value characterizes the load on the machine and it helps the user in deciding whether to use that machine or not.

Class *HPFServer* is responsible for giving locks on local machine as well as acquiring locks from remote machines. This class calculates HPF value of the local machine. *UserProgram* as well as *FTSServer* are able to query this class for HPF values.

In the current implementation HPF value is calculated by measuring the amount of time a processor is busy executing non-idle process. Lower HPF value indicates that the processor is lightly loaded. Formula[1] for calculating HPF value is given below. Let H_{n+1} be the predicted value for the next HPF value. Then, for α, $0 \leq \alpha \leq 1$,

$$H_{n+1} = \alpha c_n + (1 - \alpha) H_n$$

[1]The method is mentioned in [4] for predicting value for the next CPU burst in Shortest-Job-First CPU scheduling algorithm

Where, c_n is amount of time the processor is busy executing non-idle process at current instance of time and H_n is past history. The parameter α controls the relative weight of recent and past history in the prediction of next HPF value.

User Interface

IUser interface on *HPFServer* contains *getHPFVector* method entry as a member in it. User invokes this method on local *HPFServer* to get locks on remote machines. Every machine maintains some fixed number of locks depending on their load and capacity to service remote requests. Figure 4.7 shows the sequence diagram for acquiring locks from a user program. User program specifies the number of locks required through a parameter to *getHPFVector* method. Local *HPFServer* requests remote machines for acquiring locks until it gets as many locks as requested in presence of possible failures.

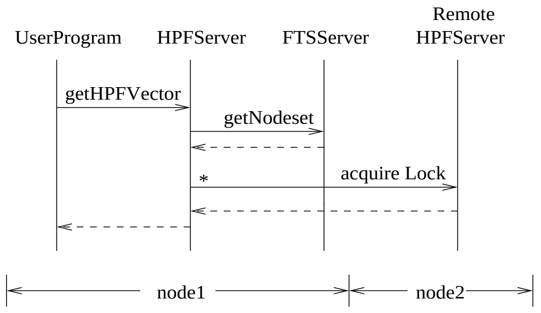


Figure 4.7: Sequence Diagram for Acquiring Locks

getHPFVector call returns the vector of *HPFValues*. User can use *IUser* interface on *HPFVector* as shown in Figure 4.6 to know how many locks are returned and can also look at individual *HPFValues* present inside the vector. User is also given interface on *HPFValue* to know the value present in it. Depending on the HPF value user might decide on task grain size to send to that machine. In general, it is desirable to send large grain tasks to lightly loaded nodes and small grain size tasks to highly loaded nodes. *HPFValue* is sent as a parameter to method *push*, which sends an ARC object to the chosen remote machine.

4.3.2 Fault Tolerance and Autoexecution Service

Fault tolerance is an important feature of any distributed system. A distributed system is said to be fault tolerant if the distributed applications can function normally in case of some node/link failures. ARC gives the user a facility to specify how system should behave in case of some failures. *AutoExecution* call on an ARC object is another facility given to the user by the system. It ensures that *Trigger* method of ARC object will be executed at least locally if it can not be executed at some remote node after following the semantics specified by the user due to failures.

Failure Detection

Failure detection is done by forming a logical ring with nodes in the system [3]. Figure 4.8 is UML activity diagram illustrating how *FTSService* is been implemented. In the logical ring each node monitors its front node and will be monitored by its back node. This monitoring is done periodically. Every node will have the same information about every other node.

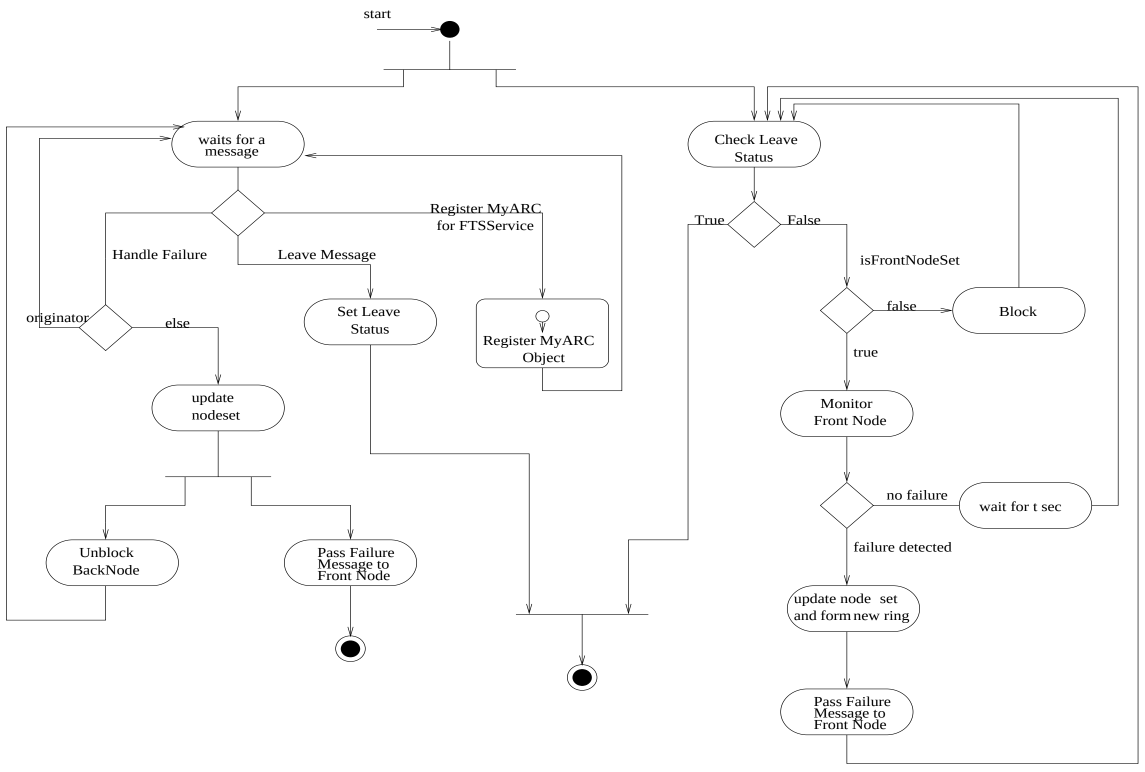


Figure 4.8: Activity Diagram for FTSService

When a failure gets detected in the system, reformation of ring structure takes place. In the ring formed before failure, the back node of the current failed node will notice the failure and changes its *nodeset* (data structure containing information about all nodes; this is same at all nodes), changes its front node pointer to point to the front node of failed node in the logical ring and back node of its current new front node to point to itself. It passes new *nodeset* information to its new front node by calling method *HandleFailure* method on front node.

When *HandleFailure* message arrives at a node, it checks whether the originator of the message is itself. If it is originator of the failure message received then that means *FailureMessage* has came back after passing through all nodes forming the ring. If it is not the originator of the failure message then it updates its *nodeset*, starts a thread to pass this message to its front node and unblocks the back node, which is waiting for return of *HandleFailure* method. Each *FailureMessage* will be assigned a version number same as originators *nodeset* version number when *FailureMessage* is prepared. Every update to nodeset information increments its version number by 1. By the time all *FailureMessages* reach their respective originators, version number of *nodeset* at every node will be equal and every node will have same knowledge of the whole distributed system. A failure message version number is meant to indicate number of failures occurred in the system.

Specifying Fault Tolerance Semantics

ARC gives the user facility to specify what semantics user wants for an ARC object in the case of failures. User has to register ARC object with the *FTSService*. Figure 4.9 shows the interaction diagram for the scenario, where user registers an ARC object for *at most k times* semantics with *AutoExecution* turned on and k=2.

As discussed previously, a node finds a failure when the node that it monitors fails or when its back node invokes *HandleFailure* method to inform about failure of anohter node. *FTSServer* then gets the list of objects that were sent to failed node by asking local *ARCSystem*. *FTSServer* checks its local registry for the action to be taken for each object. Depending on the conditions, *FTSServer* might decide one of three possible actions listed below.

1. Resend the object to some other available machine.

2. Execute locally if auto execution is set and no resends need to be done.

3. Throw an exception to the user informing about the failure.

In the scenario shown in Figure 4.9, user has registered an ARC object for *at most 2 times* semantics and set *AutoExecution* service ON. In Figure 4.9, after object is pushed by *ARCSystem* to remote machine, failure of that machine is detected. *FTSServer* gets failure notification and in cooperation with local *ARCSystem* and *HPFServer* resends the object to some available second machine. Assuming that second machine also gets failed before object returns, *FTSServer* receives failure notification. Upon receiving failure notification, in cooperation with local *ARCSystem*, *FTSServer* decides to locally execute *Trigger*.

Auto Execution

AutoExecution is another feature given to the user by ARC. User has to set this service on to make sure that *Trigger* method of the ARC object will be executed at local machine in the case of failures. If *Trigger* can not be executed on remote machine after following user specified fault tolerance semantics for an object, *FTSServer* checks for *AutoExection* status. For this purpose *FTSServer* maintains a registry of objects requiring to be executed locally.

Figure 4.10 shows the class diagram surrounding *AutoExecution* service. ARC object obtains the default implementation for *AutoExecutionON* method inside class *PContainer*.

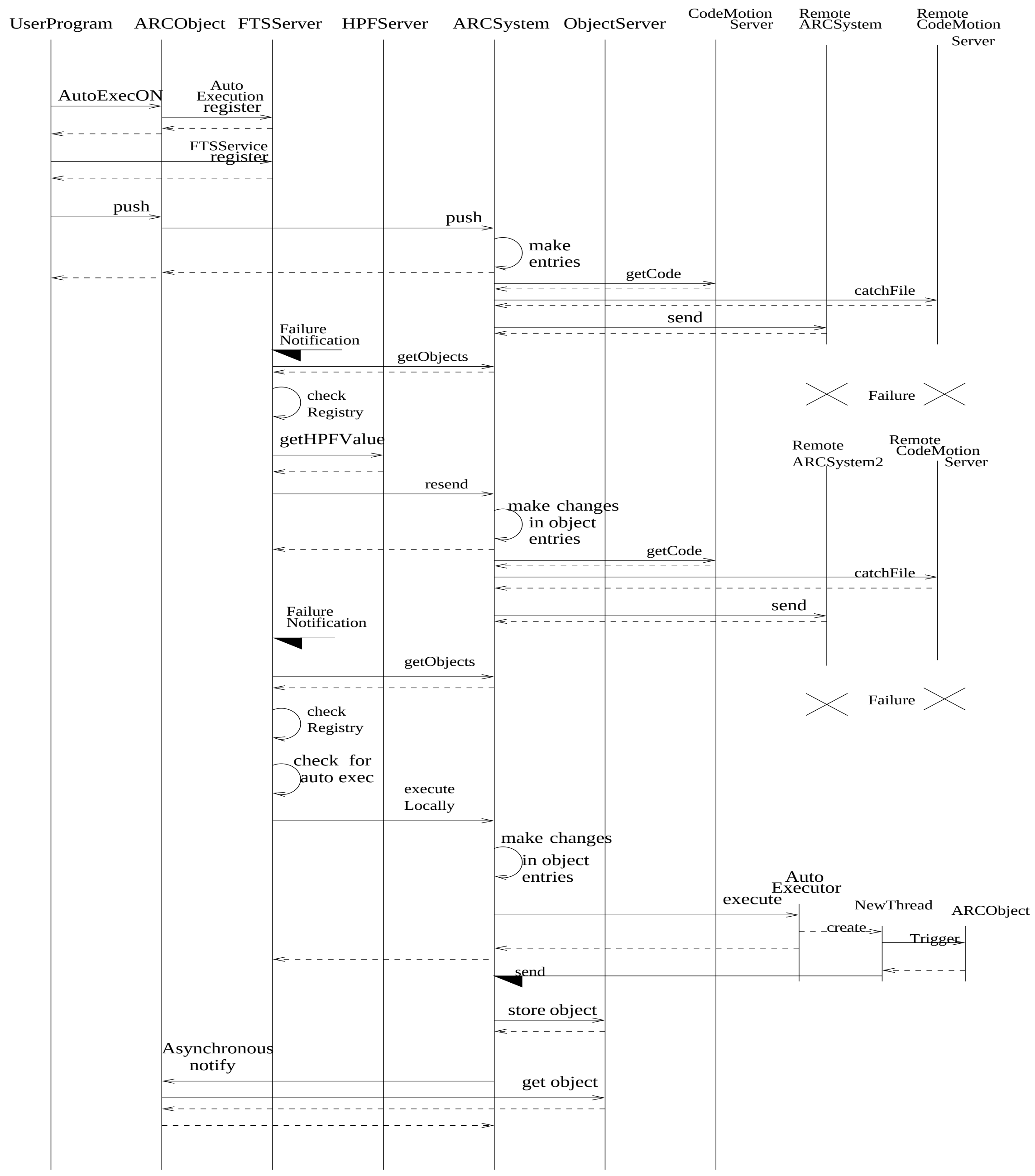


Figure 4.9: Sequence diagram when at most 2 times with Autoexecution set

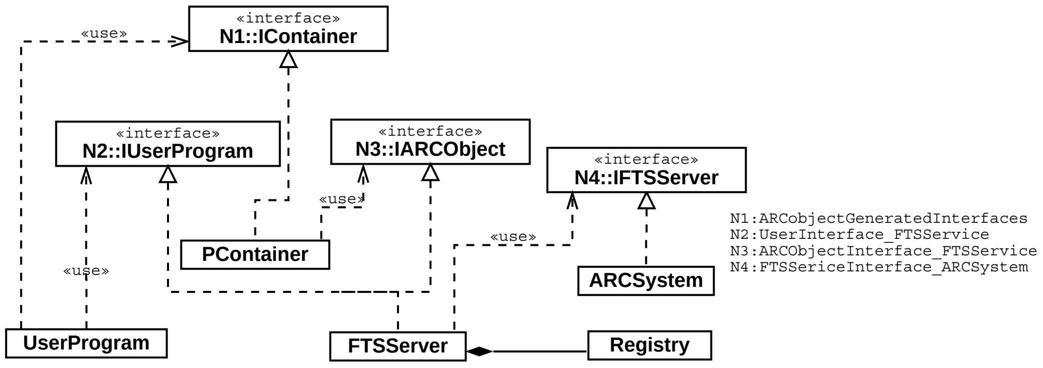


Figure 4.10: Class Diagram for AutoExecution Service

From inside this method, it calls a method on *FTSServer* using interface *IARCObject* to register for the service.

4.3.3 Object Arrival Intimation Service

Object arrival intimation service is used to export ARC object interface to the remote environment. After reaching remote machine, ARC object registers a proxy to itself with the *NamingServer*. Any program residing in remote environment can register a request at the *NameServer* for an intimation when a required object of a class arrives at that machine.

Figure 4.11 shows the class diagram for object arrival intimation service. Class *NameServer* maintains the registry of registered objects and registered requests for objects. *NameServer* notifies all the *UserPrograms* that register a request for an intimation on an object arrival.

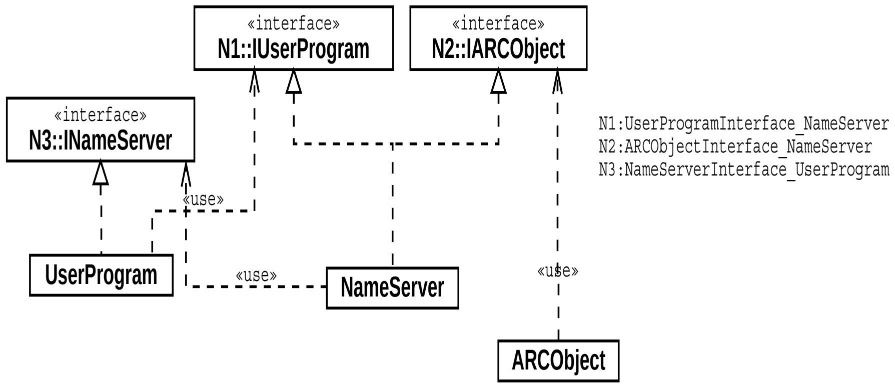


Figure 4.11: Design for Object Arrival Intimation Service

User Program Registration Interface

UserProgram uses interface *IUserProgram* to register a request with *NamingService* for an intimation of arrival of an ARC object. *UserProgram* has to realize an interface called *INameServer* in the Figure 4.11. *NameServer* class when notifying the *UserProgram* uses this interface. Any *UserProgram* increments the reference count of ARC object before executing methods on it. *UserProgram* decrements the reference count after completion of accessing the object it, which is described in the Section 3.2.1. ARC object cannot change its location if its reference count is greater than zero.

ARC Object Registration Interface

ARC object uses interface *IARCObject* realized by *NameServer* to register itself with the *NameServer*. After receiving the register request, the *NameServer* stores the request in its registry and notifies the *UserPrograms*, which have previously registered a request to get notification of the arrival of object.

4.3.4 Activation and Deactivation Services

Activation and Deactivation services are useful in working in disconnected mode of operation. A node may want to leave the ARC network after receiving ARC objects, which are programmed to be deactivated. In such a case, ARC objects may be serialized to a file on the harddisk. Later, when node rejoins the system, some user application may activate the ARC object from the serialized file on the harddisk. These two services are provided as user library, which user may reuse. Table 4.9 lists static member functions in class *Serializer*.

static members
int SerializeAndDeactivate(string filename,Object arcobject)
Serialize(string filename, Object arcobject)
Object deserialize(string filename)
void deSerializeAndActivate(string filename)

Table 4.9: Serializer Class

Figure 4.12 shows class diagram for Activation and Deactivation services. Class *Serializer* uses interface *ISerializer* on *ARCSystem* to activate and deactivate an ARC object. *UserProgram* and *ARCObject* use *Serializer* class.

Originator application that send ARC object to remote machine need to register the ARC object with *FTSService* for disconnected operation. As parameter to call *register* on *FTSService*, orinator application sends -1. This indicates to the *FTSService* that if it detects a node failure then ARC objects, which are registered for disconnected operation and sent to failed node need no action to be performed on them.

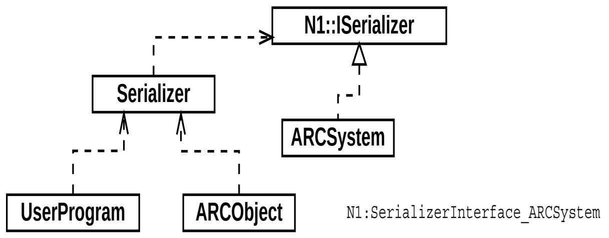


Figure 4.12: Class Diagram for Activation and Deactivation Services

4.4 Node Administrator Services

ARC addresses the problem of scalability through a dynamic join and leave protocol. In ARC a new node can dynamically join or an existing node can leave the system without affecting the functioning of other working nodes in the system.

Join Operation

Figure 4.13 shows the UML state diagram for the class *JoinServer*. This algorithm can be found in [3]. Following is the description of states of *JoinServerClass*.

1. A node wants to enter into the system first sends a *JoinRequest*.
2. The node with largest id on receiving *JoinRequest* generates a new id for the joining node and sends it as a reply; every other node sends null.
3. If no reply comes back within a timeout period then the joining node assumes that it is the first node.
4. Update *nodeset*; pass this *nodeset* information to local *FTSService* class.
5. Broadcast *Join* message.
6. Every node on receiving *Join* message updates their *nodeset* and passes the same info to their local *FTSService* classes.

Leave Operation

Figure 4.14 shows the UML sequence diagram for the *leave* operation. This algorithm can be found in [3]. Following are the sequence of steps that will take place when a node wants to leave the system.

1. Contact local *FTSService* for local node id in the distributed system.
2. Inform local *HPFService* not to give further locks to any machine.

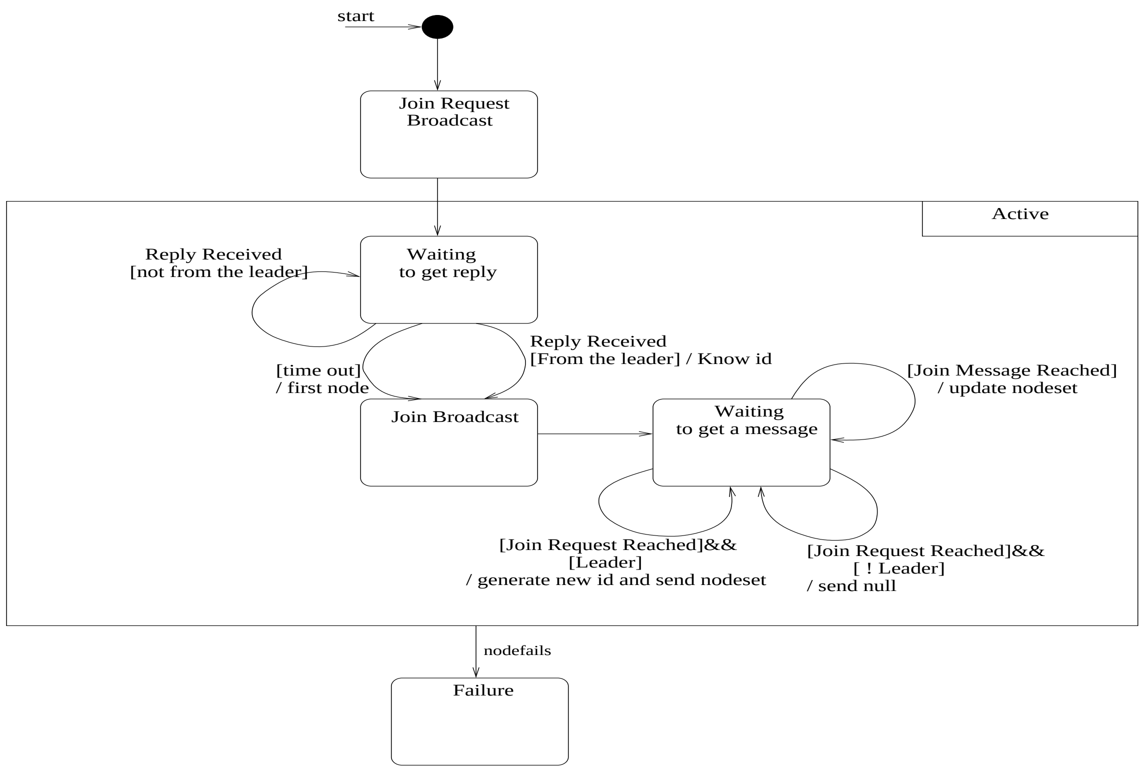

start
Join Request Broadcast
Active
Reply Received [not from the leader]
Waiting to get reply
Reply Received [From the leader] / Know id
[time out] / first node
Join Broadcast
Waiting to get a message
[Join Message Reached] / update nodeset
[Join Request Reached]&& [Leader] / generate new id and send nodeset
[Join Request Reached]&& [! Leader] / send null
nodefails
Failure

Figure 4.13: State Diagram for Join Server

3. Request *FTSService* for *nodeset* information.
4. Modify the logical ring to reflect the fact that this node is leaving.
5. Broadcast the modified *nodeset* information to all the nodes.
6. All the nodes on receiving the modified *nodeset* updates their front node and back node pointers.

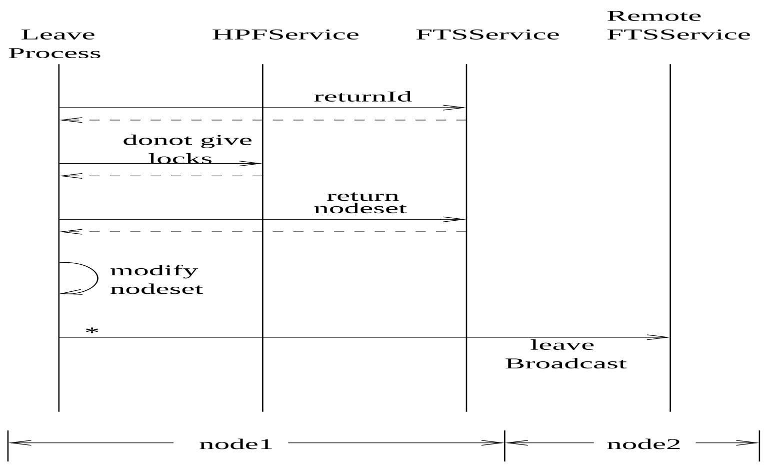


Figure 4.14: Sequence Diagram for Leave Operation

Chapter 5

Extending ARC over the Internet

Chapters 2, 3 and 4 have discussed the design and implementation of a LAN based ARC framework. This chapter presents an approach to extend ARC over the Internet. Section 5.1 presents the issues involved and objectives in extending the ARC over the Internet. Section 5.2 presents a solution approach and in Section 5.3 architectural issues of the approach are discussed. Section 5.4 presents deployment view of ARC over the Internet.

5.1 Issues and Objectives in Extending ARC over the Internet

In today's Internet age the ability to move the computation through multiple machines and performing a specific activity is beneficial in resource sharing over the Internet. It has advantages like reducing network load and autonomous execution of a specific task. Applications may work in presence of partially disconnected networks if the migrating computation is allowed to be deactivated and reactivated. Deactivation stores the state of a migrating computation to a harddisk file and reactivation restores the state of a computation from the stored harddisk file and starts the computation.

5.1.1 Issues

Main differences between LAN environment and the Internet environment are connectivity and the delay involved in communication. Broadcast algorithms that may be used in LAN environments are practically impossible for use over the Internet due to large number of machines, different configuration networks and delays in communication. Another issue of concern for ARC over the Internet is the ability to select a nearby machine if multiple machines are available while running programs to achieve speedup in execution because communication delays involved might outweigh computational speedup.

5.1.2 Objectives

The objectives in extending ARC over the Internet are two fold. Firstly, the new functionality is to be added on existing ARC implementation for LAN. Secondly, the evolution process itself follow software engineering practices.

New functionalities aimed in extending ARC are listed below

1. **ARC as mobile agent framework:** Extending migration service to allow object hopping over the Internet.

2. **Distributed application development for the Internet:** To facilitate developing distributed applications involve cooperation among processes at participant nodes available on the Internet.

3. **Platform independence:** Make all ARC services available on the Internet accessible via XML web services to make ARC platform independent. This report does not address this issue.

In terms of the software development process the objectives of extending ARC are identified below.

1. **Abstraction:** Retaining same abstraction for the user as given in ARC for the LAN, hiding internetworking details.

2. **Design and code reuse:** Reuse most of existing code of ARC for LAN.

3. **Software reengineering:** Identifying components which needs change in the implementation to work on the Internet.

5.2 A Solution Approach

To address the differences in connectivity and in the delay in communication, implementation of some components of ARC software for LAN environment has changed. For example in LAN version joining a node uses broadcast based algorithm, which may not be efficient on the Internet. Hence a special node is needed in ARC network to maintain registry of all active nodes. This special node, some times referred to as central node in this report, provides various services to participant nodes. Services provided by central node include join service, leave service, HPF service and FTS service. Communication between any two machines takes place using XML web services over HTTP. This facilitates to meet platform independence requirement. While extending the ARC framework by adding functionalities, the approach that is followed keeps the same abstraction of ARC services to the user applications written on ARC framework for both LAN and the Internet. Internet version of ARC software reuses most of the existing code of ARC written for LAN environment. Components that need change are identified from the LAN version of ARC software and modified to meet the objectives of extending ARC over the Internet.

5.3 Architecture of ARC Software over the Internet

Figure 2.2 presented architecture of ARC software for LAN. Figure 5.1 presents extended ARC software architecture to run over the Internet.

Figure 5.1 shows remote communication layer in the ARC software architecture. Remote communication layer in the architecture allows participant nodes of ARC over the Internet to interact with each other. This layer contains .NET XML webservices infrastructure, webservices and proxies to local ARC services. Web services are categorized into different components depending on the kind of service for which they act as wrapper. This layer is an incremental addition made to the architectural diagram of ARC over the LAN.

Some of the components in ARC kernel layer present in ARC software for LAN environment have also changed. In the ARC kernel level of ARC architecture shown in Figure 5.1, components are grouped as two groups depending on whether that component is same in both LAN and the Internet versions. These groups are shown using dotted rectangles inside ARC kernel layer. Components in group represented by upper dotted rectangle are same in both the versions of ARC and components in the group represented by lower dotted rectangle in ARC kernel level have different implementations in LAN and the Internet versions.

Components that need to communicate with remote node use XML webservice of that remote machine. Web services on a node typically act as wrapper programs to actual ARC service components. These webservices use proxies acquired using .NET remoting infrastructure to communicate with local ARC service components. ARC user upper level and ARC user lower level are same in both LAN and the Internet versions of ARC.

5.4 ARC Deployment View

Figure 5.2 shows deployment view of ARC software over the Internet. ARC over the Internet has participant nodes and central node as shown in the figure. A special node acts as central server providing different services to participant nodes. All participant nodes and central server are connected to the Internet. All nodes in ARC network have Internet Information Service (IIS) to allow access to webservices hosted on that machine to remote machines on the Internet. All participant nodes form a logical ring among themselves, which is shown as dotted ellipse in the figure.

Participant node and central node contain different ARC software components and provide different ARC services accessible over the Internet through HTTP and XML webservices. These XML webservices at a node act as a external interface or wrapper programs to the ARC software components at that node. Chapter 6 discusses the details of ARC software at both participant node and central node.

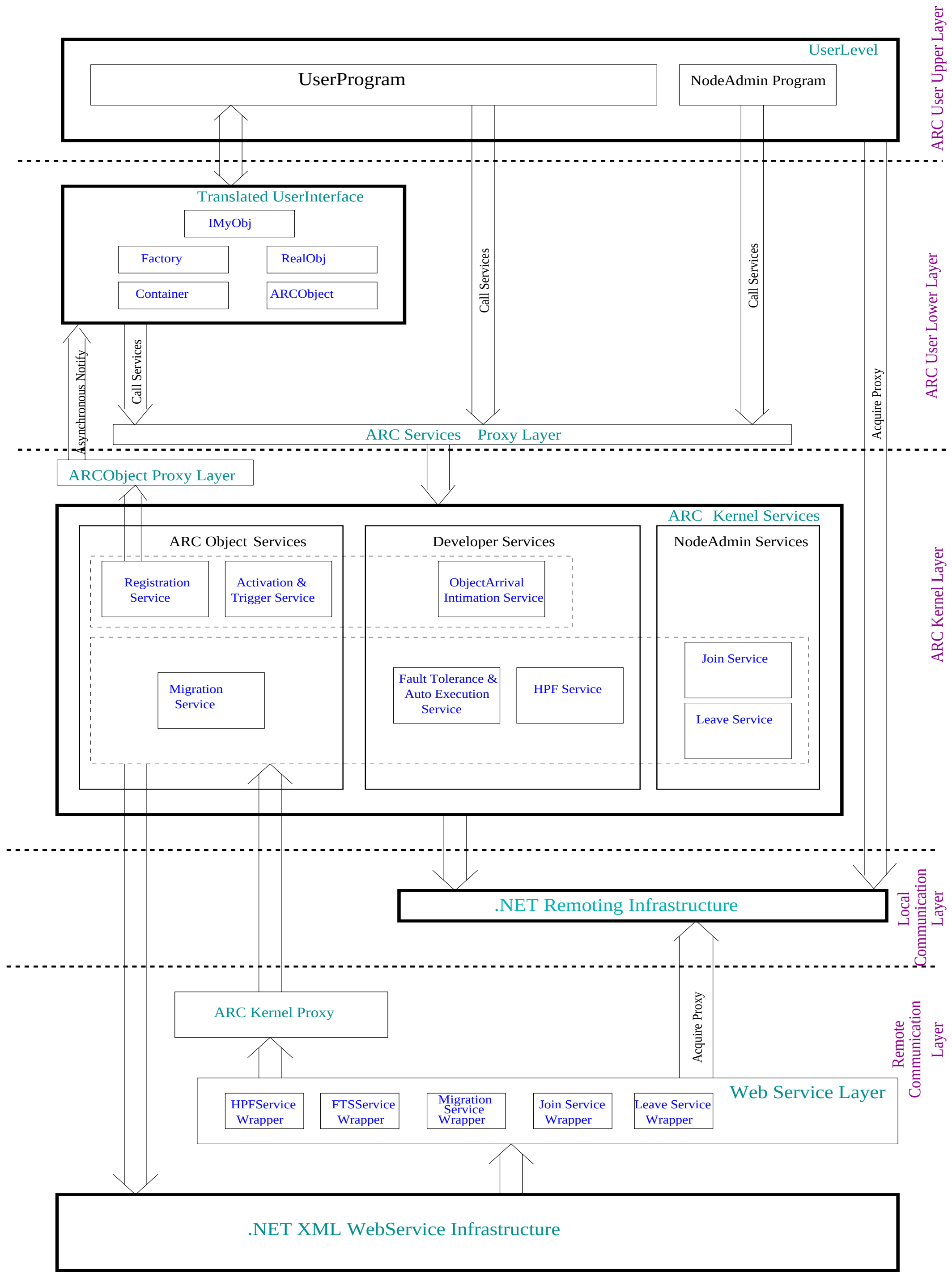


Figure 5.1: Architectural View of ARC for the Internet

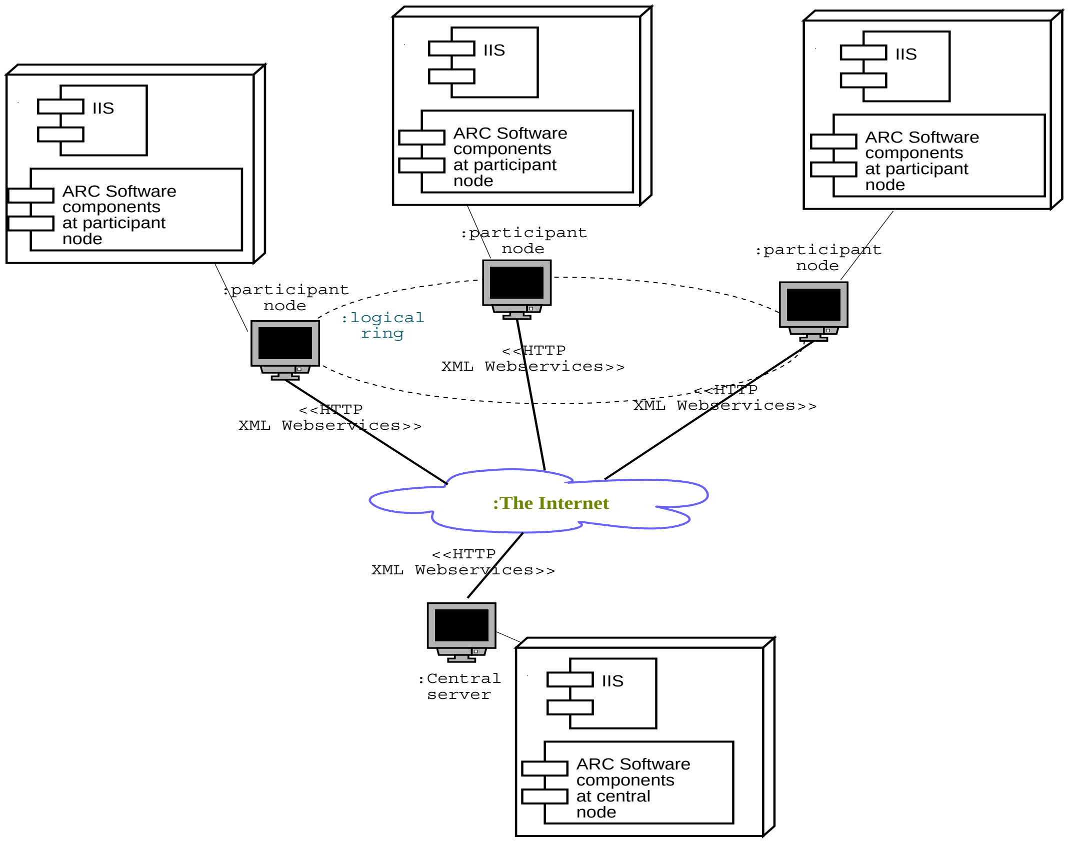


Figure 5.2: Deployment View of ARC over the Internet

Chapter 6

Design and Implementation of ARC for the Internet

Design and implementation techniques of ARC services over the Internet are presented in this chapter. Section 6.1 outlines ARC software implementation view. We follow same approach as in Chapter 4 to present ARC services classifying them according to the user of the services. Section 6.2 describes ARC object services, Section 6.3 describes developer services and Section 6.4 describes node administrator services.

6.1 Implementation View of ARC Software

ARC over the Internet has following two kinds of nodes.

1. Participant node
2. Central node

Participant nodes can dynamically join into ARC network and leave ARC network. Central node is a special node introduced in the Internet version of ARC to implement join service, FTS service and HPF service. ARC software present at these two types of nodes are not same. Section 6.1.1 shows implementation view of ARC software written for participant nodes in ARC network. Section 6.1.2 presents implementation view of ARC software written for central node.

6.1.1 ARC Software at Participant Node

ARC software at participant nodes is comparable with the ARC software over the LAN. Table 6.1 summarizes the differences in ARC software over LAN and the ARC software at participant node over the Internet. Entries in modified components column of table list all the components that have changed from LAN to the Internet extension. The table also identifies unmodified components and new components introduced in the Internet version of ARC software.

Modified Components	Fully Reused Components	New Components
ARCSystem	Object Server	PARCSystemWrapper
HPFServer	Code Motion Server	PHPFServiceWrapper
FTSServer	NameServer	PFTSServiceWrapper
JoinServer	Startup Program	
	Serializer	

Table 6.1: Comparison between the LAN and the Internet version of ARC software

Figure 6.1 shows component diagram for ARC software developed for a participant node. To avoid mixing of dependency lines, component *Startup Program* has not been shown in Figure 6.1. Its functionalities are exactly same as shown in Figure 4.1. In the remaining sections a description of newly introduced components is given. Prefix 'P' is used for participant node component names, prefix 'C' is used for central node component names and the rest of the component names are for participant node components.

PARCSystemWrapper

This component is used by remote *ARCSystem* components. It is responsible for acting as a wrapper to local *ARCSystem*. It is accessible over the Internet to other nodes in the system. Functionalities of *ARCSystemWrapper* are listed in Table 6.2. These functionalities help in transferring object and code from one node to another over the Internet. This component is shown as Migration Service Wrapper in Remote Communication Layer in Figure 5.1.

Web Services
void ReceiveMessage(WrapperMessage message)
void ReceiveAsyncReturnMessage(AsyncReturnMessageClass message)

Table 6.2: ARC System Wrapper

PHPFServerWrapper

This component is used by remote *HPFServer*. This component is responsible for acting as a wrapper to local *HPFServer*. It is accessible over the Internet to other nodes in the system. Functionalities of *HPFServerWrapper* are listed in Table 6.3. *HPFServer* acquires locks on remote node by invoking web services published by this component.

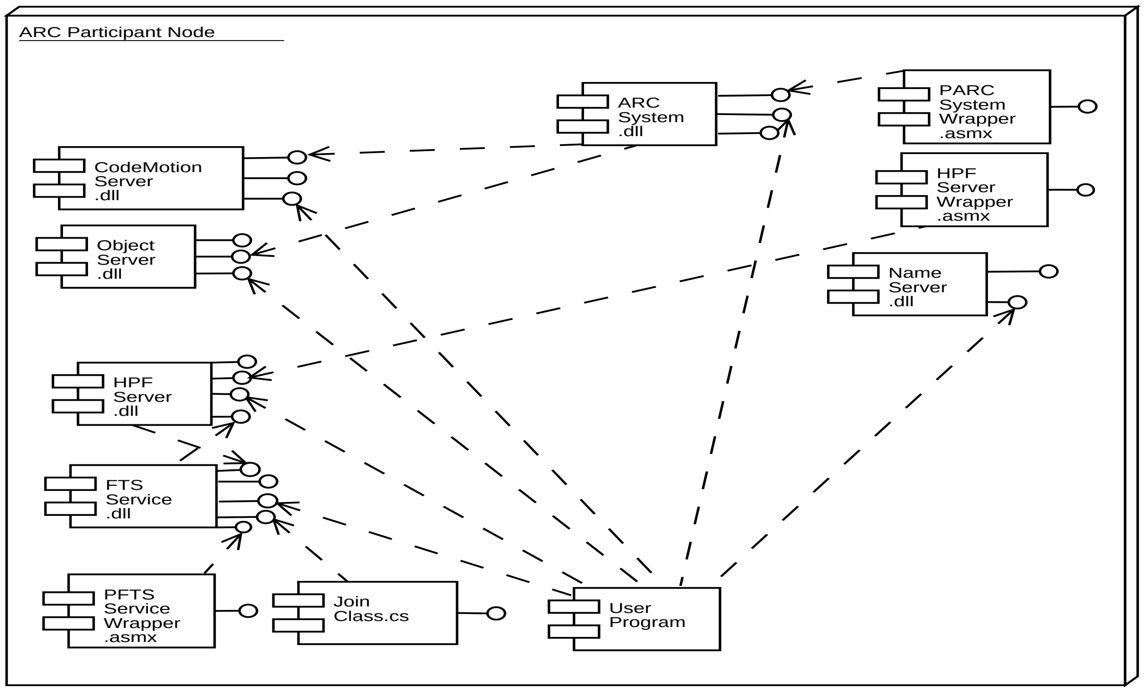


Figure 6.1: Component Diagram of ARC Software Present at a Participant Node

Web Services
HPFValue acquireLock()

Table 6.3: HPF Server Wrapper

PFTSServiceWrapper

This component is used by remote *FTSServer* and central node. This component is responsible for acting as a wrapper to local *FTSServer*. It is accessible over the Internet to other nodes in the system. Functionalities of *FTSServiceWrapper* are listed in Table 6.4. *FTSServer* of a node monitors front node of that node in ARC ring structure by calling a webservice of *PFTSServiceWrapper* at front node. Central node uses web services published by *PFTSServiceWrapper* in reforming the ARC ring structure in case of node failures.

6.1.2 ARC Software at Central Node

Central node is a special node introduced in ARC over the Internet. It implements join service, FTS service and HPF service. Figure 6.2 shows component diagram for ARC software at central node.

Web Services
int LiveCheck()
void FailureNotification(NodeId failednode)
void ChangeFrontNode(NodeId nodeid)

Table 6.4: FTS Service Wrapper

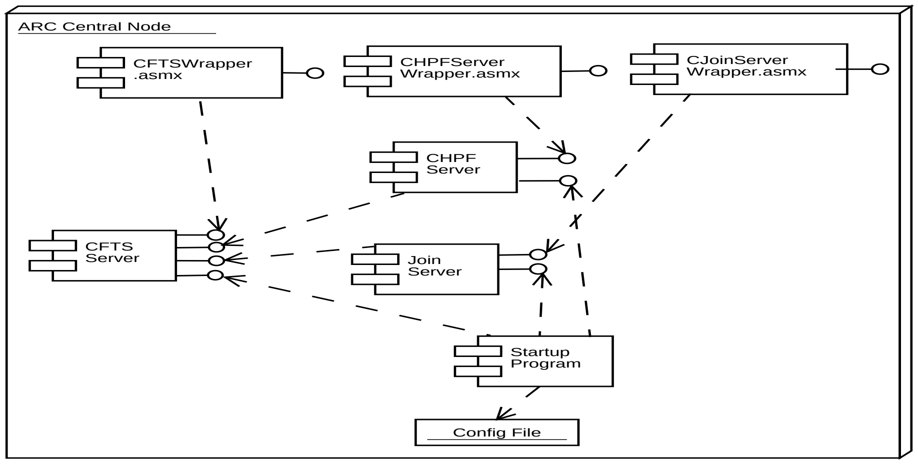


Figure 6.2: Component Diagram of ARC Software Present at a Central Node

Table 6.5 categorizes components at central node based on their visibility to remote nodes over the Internet. Wrapper components implement web services, which are hosted on the Internet using Internet Information Service (IIS). Remaining components that are not visible on the Internet implement ARC services. These components cannot be accessed directly from remote machines and are accessed through wrapper components. Description of these components is given below.

Visible Components	Components Not Visible to the Internet
CFTSServerWrapper	CFTSServer
CHPFServerWrapper	CHPFServer
CJoinServerWrapper	CJoinServer
	Startup Program

Table 6.5: Visibility of Components at Central Node over the Internet

CFTSServer

This component is used by *CFTSServerWrapper*, *CHPFServer* and *CJoinServer*. Interfaces of this component are listed in Table 6.6. *CJoinServerWrapper* uses interface *IJoin* to make changes in ARC ring structure when new nodes join into the system. *CFTSWrapper* uses the interface *ICFTSServerWrapper* on *CFTSServer* to reform the ARC ring structure in case of failures. *CHPFServer* uses interface *ICHPFServer* in implementing lock acquisition.

ICJoin
void insertNewNode(NodeId newnode);
NodeId frontNodeOf(NodeId nodeid);
ICFTSServerWrapper
void HandleFailure(NodeId failednode)
ICHPFServer
ArrayList getLiveNodes();

Table 6.6: Central FTS Service Component

CHPFServer

This component is used by *CHPFServerWrapper*. It handles lock acquisition and lock release. Table 6.8 lists interfaces of this component.

ICHPFServerWrapper
UserInterface_HPFVector.IUser getHPFVector(int size)
UserInterface_HPFValue.IUser getHPFValue(string machaddr)
int Release(Object hpfvalue)

Table 6.7: Central FTS Service Component

CJoinServer

This component is used by *IJoinServerWrapper*. In collaboration with *CFTSServer*, it allows new nodes to register in the ARC ring.

IJoinServerWrapper
JoinResponse RegisterJoin(string ip, bool registerforNotifyalso)

Table 6.8: Central FTS Service Component

Startup Program

Start up program is responsible for starting all server components. *StartupProgram* uses a configuration file to read configuration setting of the user. It uses this information while starting all the components. It starts various services one by one as per their dependencies and invokes the initialization member functions on them.

CFTSServerWrapper

This component is used by participant node *FTSServer*. This component is responsible for acting as a wrapper to local *CFTSServer*. It is accessible over the Internet to other nodes in the system. Functionalities of *CFTSServiceWrapper* are listed in Table 6.9.

Web Services
void HandleFailure(NodeId failednode)

Table 6.9: Central FTS Service Wrapper

CHPFServerWrapper

This component is used by participant node *HPFServer*. This component is responsible for acting as a wrapper to local *CHPFServer*. It is accessible over the Internet to other nodes in the system. Functionalities of *CHPFServerWrapper* are listed in Table 6.10. Structure *HPFValue* is a type <ip-address, lock-no, actual hpf value>.

Web Services
HPFVector getHPFVector(int size)
HPFValue getHPFValue(string ip)
int Release(Object hpfvalue)

Table 6.10: Central HPF Server Wrapper

CJoinServerWrapper

This component is used by participant node *JoinServer*. This is responsible for acting as a wrapper to local *CJoinServer*. It is accessible over the Internet to other nodes in the system. Functionalities of *CJoinServerWrapper* are listed in Table 6.11. Structure *JoinResponse* contains the id that is allocated the newly joining node and the id of the current front node, which is to be monitored by the newly joining node.

Web Services
JoinResponse Register(string ip, bool registerforNotifyalso)

Table 6.11: Central Join Server Wrapper

6.2 ARC Object Services

Section 4.2 discussed ARC object services for the ARC over LAN in detail. Table 6.12 categorizes ARC object services based on the need to communicate with some remote node on the network to implement the service. Implementation of services that don't require communication with remote nodes is same in both LAN version of ARC and the Internet version. Services which involve in communication with remote nodes are described in the remaining section.

Locally implemented constructs	Constructs requiring communication with remote nodes
Registration	Push
Sync	Connect
	Retraction
	Multiple-hopping

Table 6.12: ARC object services based on need to communicate with remote nodes

6.2.1 Migration Service

Migration service deals with sending an ARC object to remote machines, executing method *Trigger*, and synchronizing the state of the migrated object at original sender application. Construct *push* is used to send an ARC object to remote machines and construct *Sync* is used for synchronization. Implementation of construct *push* uses ASP.NET webservices for sending the code and state of the ARC object. Implementation of *Sync* doesn't require any communication with remote nodes and it is same as in LAN version of ARC described in Section 4.2.2.

Push Operation

Figure 6.3 shows sequence diagram for *push* operation in the Internet version. While sending an ARC object to remote machine, *ARCSystem* uses web service wrapper of destination machine. Wrapper at remote machine passes received object and code to *ARCSystem*.

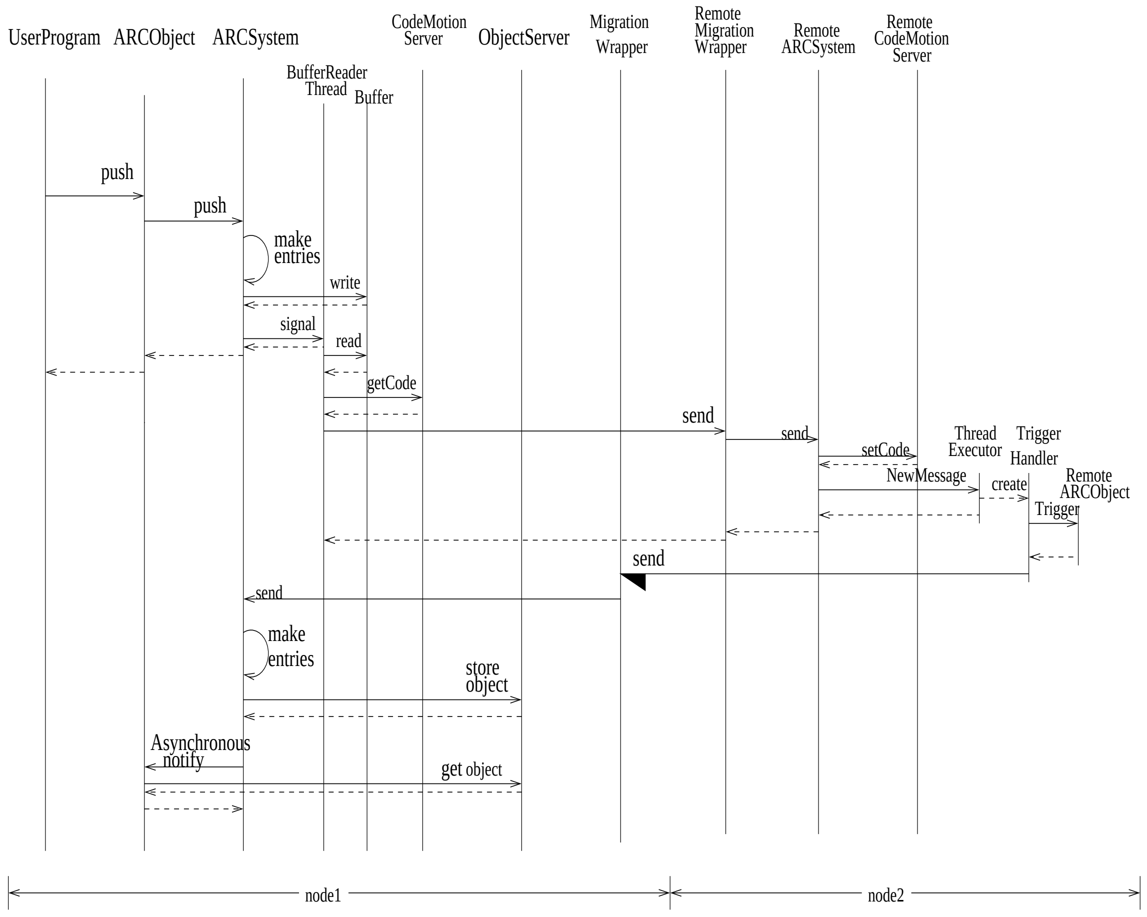


Figure 6.3: Sequence Diagram for Nonblocking Push Operation

Asynchronous Return

ARCSystem receives the asynchronous arrival of migrated ARC object, through web service wrapper at the local node. Remote node invokes a method on wrapper to return object.

6.2.2 Object Hopping

ARC services to support object hopping are *GracefulRetract*, *connect* and multi-hopping. Implementation of construct *GracefulRetract* is same in LAN version of ARC and the Internet version of ARC. Description of *GracefulConstruct* can be found in Section 4.2.3. Implementation of construct *connect* has changed to use ASP.NET webservices in the Internet version. Implementation of multi-hopping also uses ASP.NET webservices to send an ARC object through multiple machines.

Connect Construct

Implementation of *connect* construct uses ASP.NET webservices. User has to write webservices code as wrapper to ARC object at remote node. Webservice code is transmitted to remote node along with ARC object code while pushing the ARC object to any remote node. At remote node webservice code is published for remote access by copying it into IIS virtual directory.

Figure 6.4 shows sequence diagram for *connect* construct. When method *connect* is called, a proxy to the webservice is created and its URL is assigned to ARC object's current location. Subsequent local method calls on ARC object are delegated to remote ARC object through proxies.

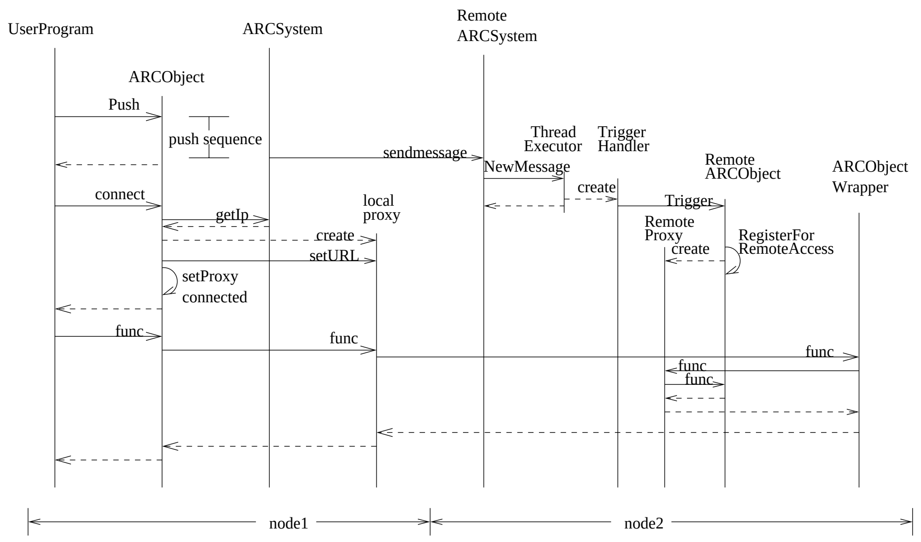


Figure 6.4: Sequence Diagram Showing Working of Connect Construct

6.3 Developer Services

ARC developer services include HPF service, FTS service, Auto Execution service, Object Arrival Intimation service and, Activation and Deactivation services. Section 4.3 discussed ARC developer services in LAN environment. Table 6.13 categorizes ARC developer services based on the need to communicate with some remote node on the network to implement the service. Implementation of services that doesn't require communication with remote nodes is same in both LAN version of ARC and the Internet version. Services which communicate with remote nodes are described in the remaining section.

Locally implemented services	Services requiring communication with remote nodes
Object Arrival Intimation Service	HPF Service
Activation and Deactivation Service	FTS Service
Auto Execution Service	
Activation and Deactivation Service	

Table 6.13: ARC object services based on need to communicate with remote nodes

6.3.1 HPF Service

HPF service includes providing user interface to get HPF value of remote machines and acquiring locks to send ARC objects. This section presents processor selection and corresponding user interface given by HPF service to the distributed programs developer. Subsequently HPF service protocol used to acquire locks from remote machines in the Internet version of ARC is presented.

Processor Selection

Figure 6.5 shows sequence diagram of acquiring a lock on remote machine. User makes a request to get HPF value on local HPF server. Local HPF server passes the request to central node, which requests other active participant nodes for locks and returns to the requesting HPF server hence to the calling user program. User interface in both versions of ARC are the same.

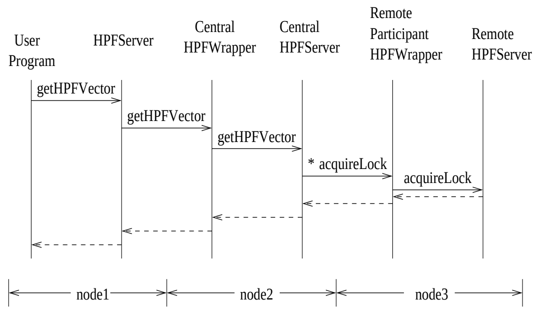


Figure 6.5: Sequence Diagram showing Acquiring Locks

HPF Service Protocol

Figure 6.6 shows class diagram for HPF service at central node. Central node HPF sever uses participant node's HPF service wrapper to acquire locks. *CHPFServiceClass* also uses central node FTS service to know about active participant nodes.

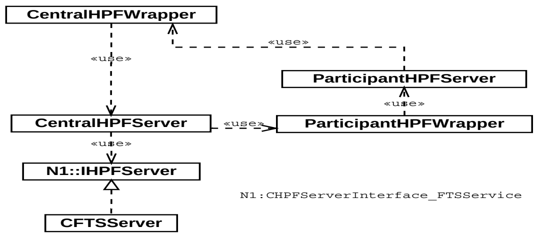


Figure 6.6: Class diagram for HPFService at Central Node

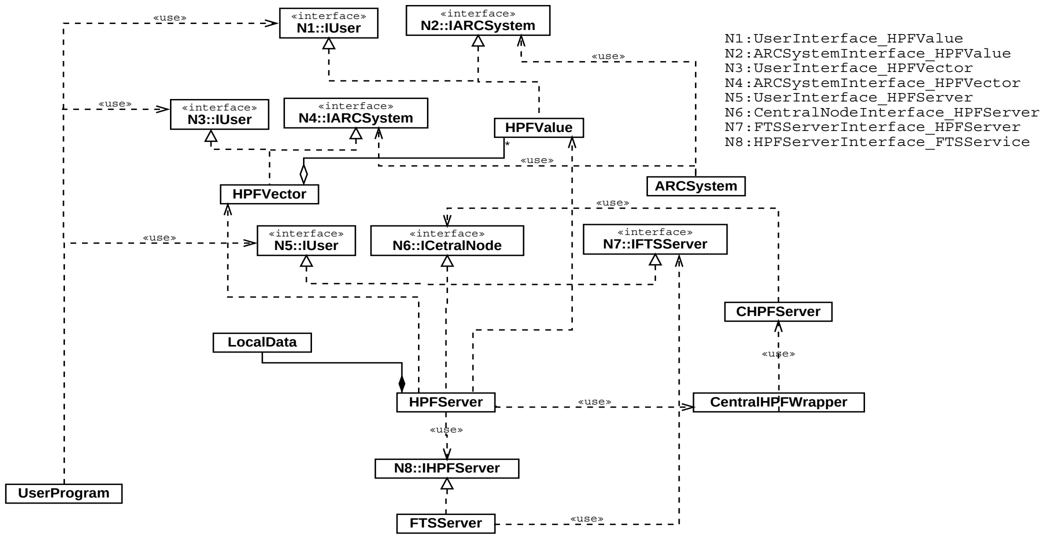

«interface» N1::IUser
«interface» N2::IARCSystem
N1:UserInterface_HPFValue
N2:ARCSystemInterface_HPFValue
N3:UserInterface_HPFVector
N4:ARCSystemInterface_HPFVector
N5:UserInterface_HPFServer
N6:CentralNodeInterface_HPFServer
N7:FTSServerInterface_HPFServer
N8:HPFServerInterface_FTSService
«interface» N3::IUser
«interface» N4::IARCSystem
HPFValue
HPFVector
ARCSystem
«interface» N5::IUser
«interface» N6::ICetralNode
«interface» N7::IFTSServer
CHPFServer
LocalData
HPFServer
CentralHPFWrapper
N8::IHPFServer
UserProgram
FTSServer
«use»

Figure 6.7: Class diagram for HPFService at Participant Node

Figure 6.7 shows class diagram for HPF service at participant nodes of ARC network. HPF server at participant node uses webservices published by *CentralHPFWrapper* to request central node for locks on remote participant nodes. Central node in turn uses interface *ICentralHPFServer* on *HPFServer* of participant node to acquire locks on behalf of requesting participant node.

6.3.2 Fault Tolerance

Section 4.3.2 described fault tolerance service of ARC over the LAN. In the Internet version, ARC central node maintains the ARC ring structure. Participant nodes inform central node upon detecting a failure. Following is the description of failure detection method used in the Internet version of ARC.

Failure Detection

Failure detection is done by forming a logical ring with participating nodes in the system. Each node in the logical ring monitors its front node and it is monitored by its back node. A participant node knows only about its front node. It has no knowledge about any other participant node in the system.

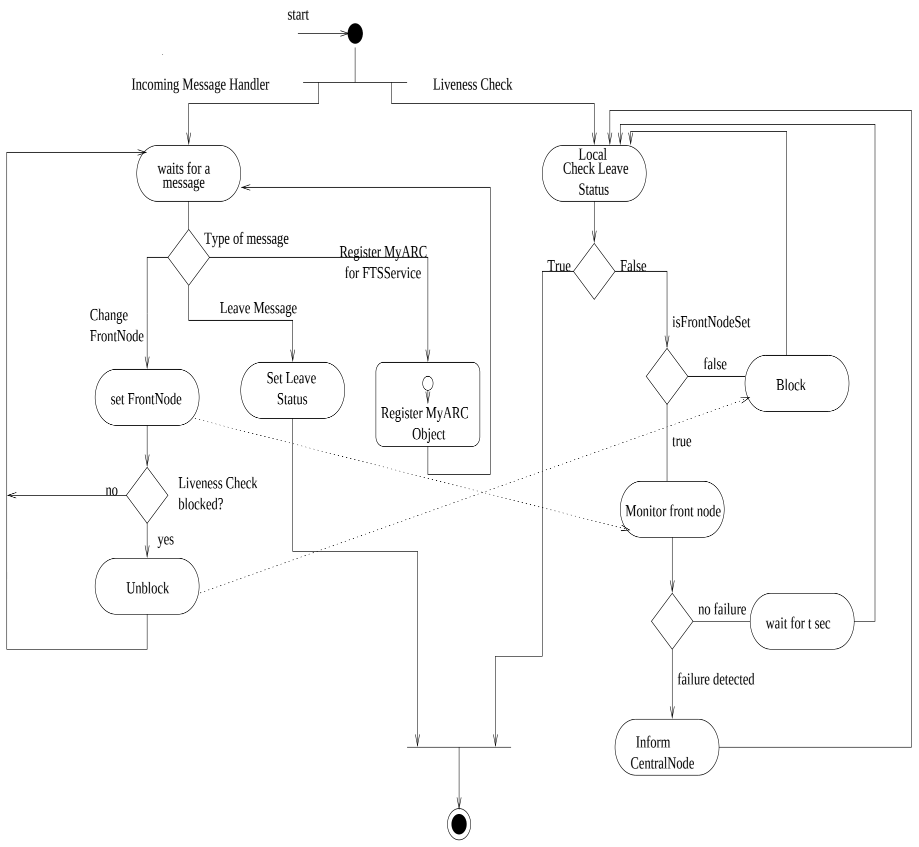


Figure 6.8: Activity Diagram for FTSService on Participant

Figure 6.8 is UML activity diagram illustrating implementation of *FTSService* in the Internet version. A participant node upon detecting a failure of its front node in the logical ring passes the failure message to central node. Central node reforms the ring by deleting failed node's entry from the ring structure. Central node finds new front node for the participant node, which has detected a failure, and calls a webservice on it to update its front node information. Central node notifies all active participant nodes about the failure. Upon receiving a failure notification, *FTSServer* at participant nodes implement fault tolerance semantics specified by user applications for ARC objects. *FTSServer* uses *ARCSystem* to know about ARC objects that were sent to the failed node.

Specifying Fault Tolerance Semantics

Figure 6.9 shows the interaction diagram for the scenario, where user registers an ARC object for *at most k times* semantics with *AutoExecution* turned on and k=2. Sequence of messages is similar to sequence of messages shown in Figure 4.9 for the LAN environment. In the Internet version, communication between any two machines takes place through webservices published by wrapper components.

6.4 Node Administrator Service

In ARC over the Internet any node can dynamically join or leave the ARC network. Section 4.4 discussed node administrator services of ARC over the LAN environment. In the Internet version of ARC, a participant node uses bootstrapping node rather than broadcasting algorithms as in LAN version of ARC to join or leave the system. Bootstrapping node, which is referred to as central node through out the report is responsible for allowing participant nodes to join and leave the system. The join and leave sequence is discussed below. Front node is the node that is monitored by the newly joining node and back node monitors the newly joining node. Appropriate monitoring configuration may be programmed in *CentralFTSServer*.

Join Operation

Figure 6.10 shows the UML sequence diagram of join operation. Newly joining node has to know the location of central node server beforehand. Central node allocates a new id to the newly joining node. It returns the allocated node id and front node information as response to the join request. Following are the steps that take place when a node wants to join the system.

1. A node wants to enter into the system first sends a *JoinRegister* request to central node.
2. Central node in co operation with central FTSServer assigns a new id to requesting node.
3. Central FTSServer makes changes in ring structure.
4. Central FTSServer informs a node about change in front node depending on whether its current front node has changed due to new join request.
5. Central JoinServer returns new id and information about front node to the requesting node.

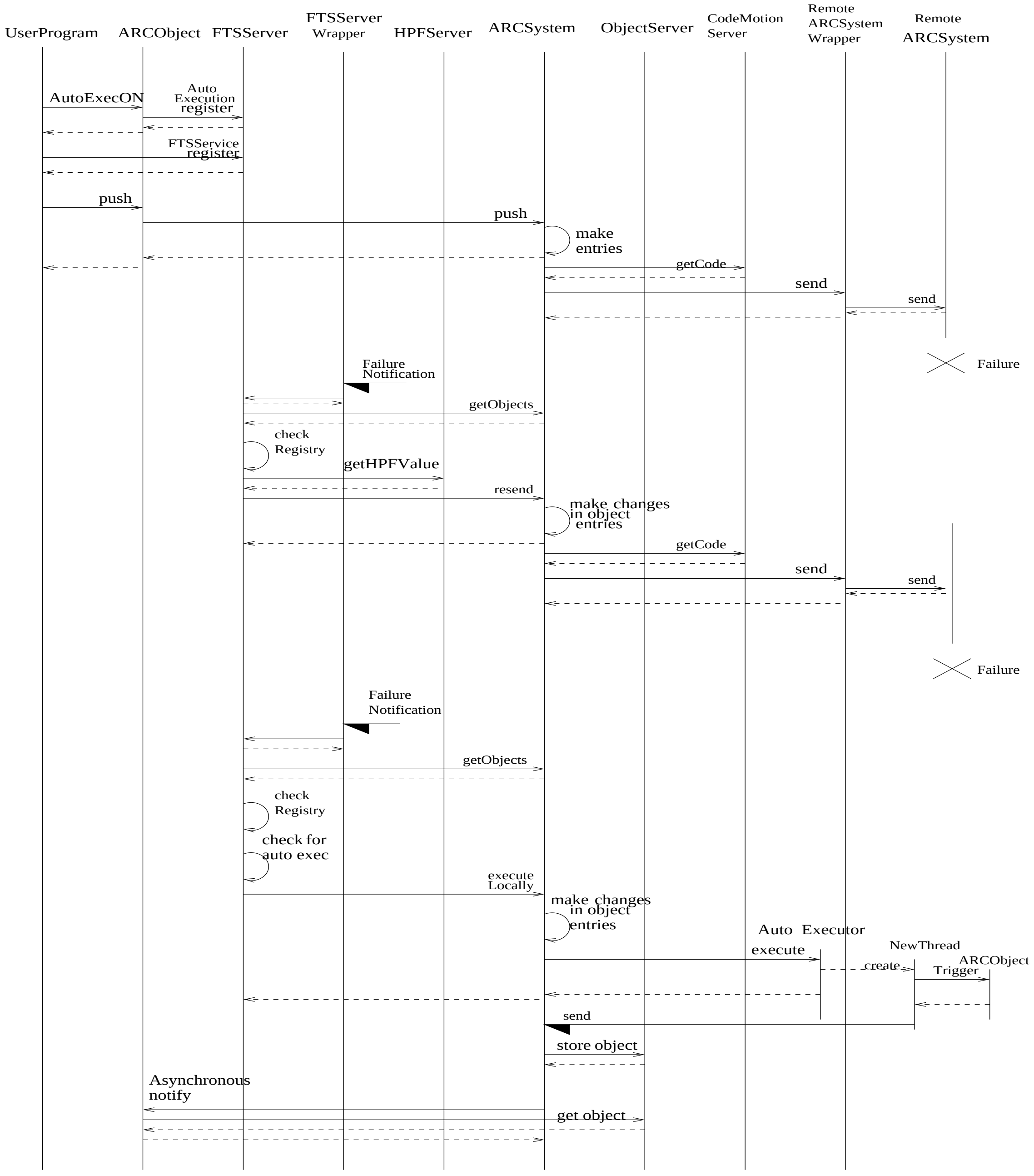


Figure 6.9: Sequence diagram when at most 2 times with Autoexecution set

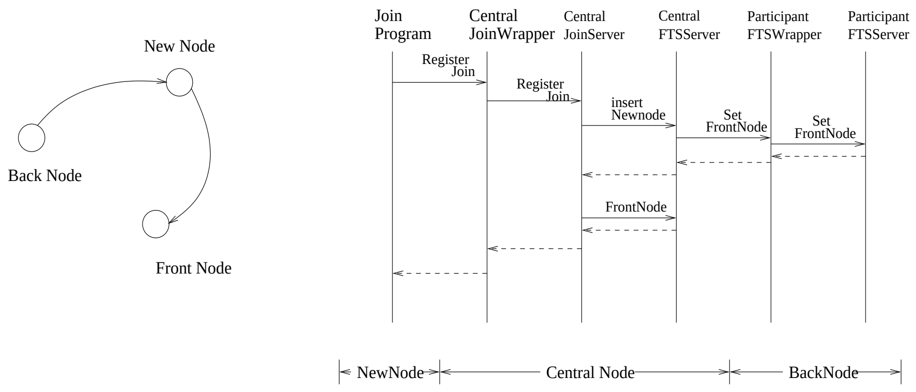


Figure 6.10: Sequence diagram for Join operation

Leave Operation

Figure 6.11 shows UML sequence diagram of leave operation. Following are the steps that take place when a node wants to leave the system.

1. Contact local *FTSServer* for local node id.
2. Inform local *HPFServer* not to give further locks to any machine.
3. Send leave message to central node
4. Central node *CJoinServer* removes leaving node id and invokes method *HandleLeave* on local *CFTSServer*
5. *CFTSServer* calls method *ChangeFrontNode* of a participant node if that participant node's front node changes due to current leave request.
6. *CFTSServer* informs all active participant nodes about the leave.

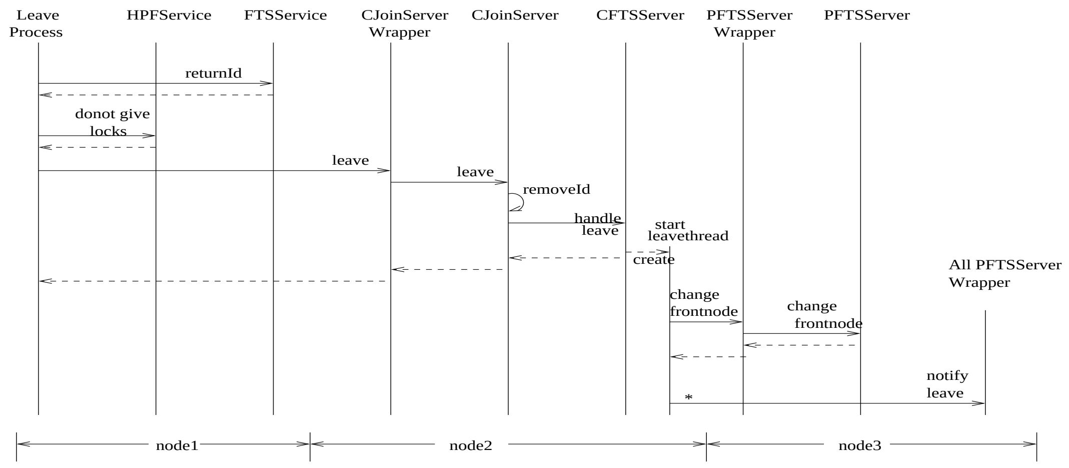


Figure 6.11: Sequence Diagram for Leave Operation

Chapter 7

Example Applications

High level ARC program constructs have been discussed earlier in Sections 4.2.2, 4.2.3, 4.3.1, 4.3.2, 4.3.3, 4.3.4, and 4.4. This chapter presents three example applications to demonstrate programming using these constructs. Section 7.1 explains a *copy paste application* that involves executing methods at remote node from the originator machine of ARC object. Section 7.2 presents *workshop organization application*, which involves exporting ARC object interface to the remote environment after reaching remote machine. Section 7.3 demonstrates a method to write applications, which operate in disconnect mode.

As described in Section 3.1, application development process using ARC involves following steps.

- ARC object interface specification
- Implementation of *Real* class.
- Use of implemented ARC object inside an application.

This Chapter presents example applications by describing above mentioned steps for the applications.

7.1 Distributed Copy Paste Application

Distributed copy paste application is useful in copying text at one machine and pasting it at another machine. An ARC object stores the text and acts as messenger between originator and remote machine. ARC object migrates to a remote machine and registers a proxy to enable access to its interface from remote environments. Remote machine environment extracts the text from it and displays the text on a form. Originator machine uses *connect* construct to subsequently read and modify the copied text. Every machine runs a *copy paste* daemon, which registers a request with *NameServer* at startup for an intimation of arrival of *CopyPasteARCObject*. Text may be pasted and retrieved from this daemon. This daemon is responsible for using ARC framework for transmitting text. Various components of the application are described below.

7.1.1 Interface Specification

Following interface is used in copy paste application to create an ARC object.

```
using System;
using InterfaceToUI;
namespace CPObjectInterface
{
        public interface ICP
        {
                Object getData();
                void setData(Object data);
                void setDataAndNotify(Object data);
                void RegisterApp(IUI app);
        }
}
```

This interface is used in creating *Generated Interfaces*, specified in Section 3.2. User applications at both ARC object originator machine and remote machine use *Generated Interfaces* to interact with ARC object.

7.1.2 Implementation of the Interface

Interfaces *ICP* and *ITrigger* are implemented in class *Real*. A proxy to the ARC object is created inside *Trigger* method and it is registered to allow access to it from remote environments. *Trigger* method also registers the proxy at the *NameServer*. *NameServer* notifies all applications, which have registered a request for intimation of arrival of *CopyPasteARCObject*. Code for class *Real* is shown below.

```
 public class Real:PReal,CPObjectInterface.ICP,ITrigger {
   private Object Lock = new Object();
   public void Trigger(){
     // 1. create and register a proxy
     RegisterForRemoteAccess();
     // 2. register with nameserver
     ARCObjectInterface_NamingService.IARCObject nameserver = null;
     if(this.Proxy != null){
         Object proxy = this.Proxy;
         nameserver = (ARCObjectInterface_NamingService.IARCObject)
                           Activator.GetObject(typeof(
                           ARCObjectInterface_NamingService.IARCObject),
                                "tcp://localhost:9123/NameServerClass");
         nameserver.Register("CopyPasteARCObject",proxy);
      }
      // 3. block
      Monitor.Enter(Lock);
      Monitor.Wait(Lock);
      Monitor.Exit(Lock);
      // 4. unregister proxy
      UnRegisterForRemoteAccess();
      // 5. unregister with nameserver
      if(nameserver != null)
         nameserver.UnRegister("CopyPasteARCObject");
   } // end of Trigger method
   public void setData(Object dat){
        // implementation
   }
   public void setDataAndNotify(Object dat){
        // implementation
   }
   public Object getData(){
        // implementation
   }
   public void RegisterApp(IUI app){
        // implementation
   }
   public void OnReturn(){
        // implementation
   }
   public void OnRetract(){
        // implementation
   }
} //end of class definition
```

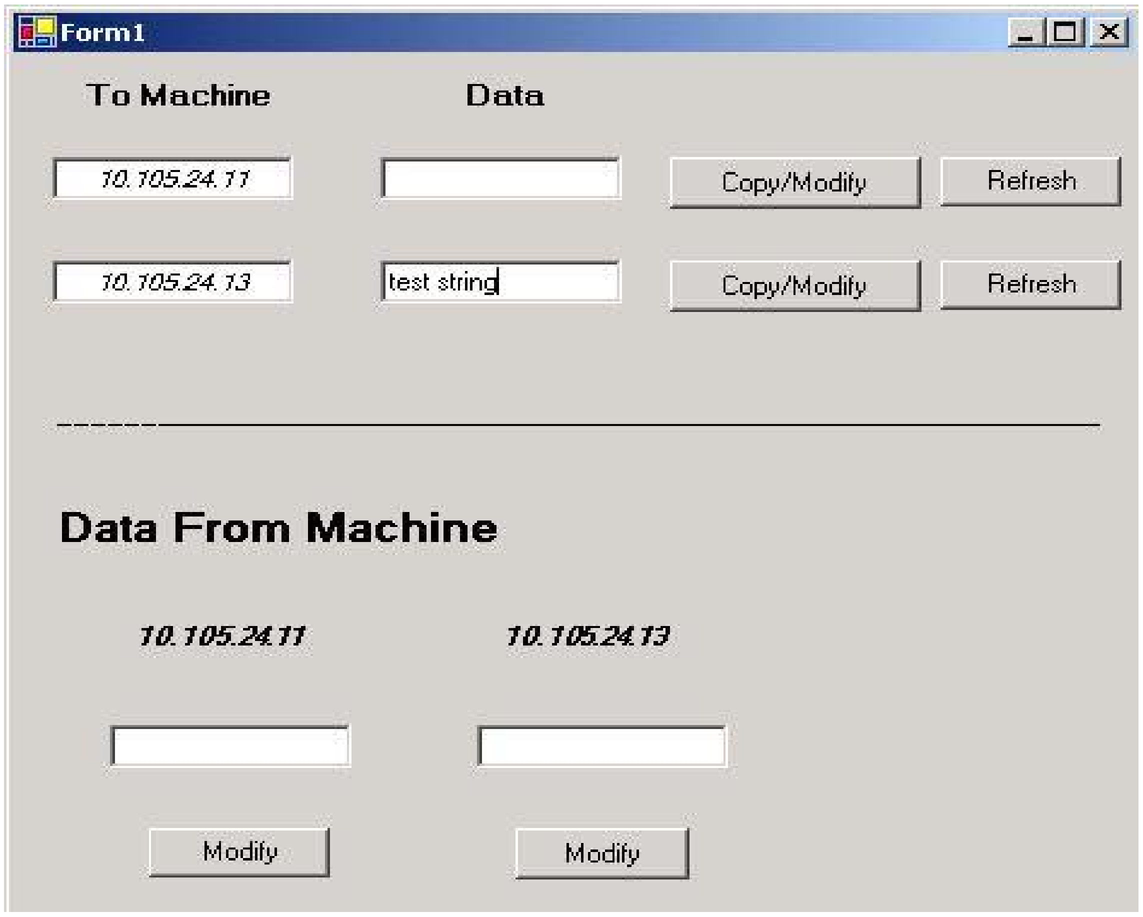
7.1.3 GUI of Distributed Copy Paste Application

Figure 7.1 shows the GUI of *Copy Paste Application.* The application registers a request for *CopyPasteARCObject* with *NameServer*. It gets the notification, when *CopyPasteARCObject* registers a proxy with the *NameServer* through a call to method *register* on *NameServer* from method *Trigger*.

Figure 7.1 has string, "test string" in one of the Textboxes. When button labeled Copy/Modify located immediately next to "test string" is clicked for the first time, an ARC object is instantiated. String in the Textbox is stored into the ARC object and sent to the destination machine, in this case, address of destination machine is *10.105.152.13*.

ARC object creates a proxy to itself and registers it with the *NameServer* from inside method *Trigger* method. *NameServer* then notifies all the registered applications.

Remote *Copy Paste Application* gets the notification from *NameServer*. It extracts the string and sender machine IP to display the message string.



10.105.24.11

10.105.24.13

test string

10.105.24.11

10.105.24.13

Figure 7.1: Copy Paste Application Interface

Proxy to the object can be acquired from sender machine by calling method *connect*. Clicking on *Refresh* button at originator machine makes corresponding Textbox to show current value of string of ARC object. Any modifications to the string data from original sender machine requires a click on *Copy/Modify* button and any modification to the string data from remote environment requires a click on *Modify* button.

7.2 Workshop Organization Application

This section presents an application to organize workshops. In a simplified scenario, it is assumed that a seminar room and refreshments from canteen are needed to organize a work shop. *Workshop Organization Application* uses ARC objects to book the seminar room and to place orders with a canteen. Two different ARC objects carry the requests on behalf of the organizer, one for seminar room and other one for refreshments.

System may be automated by making required information like availability of seminar room available to the incoming object at machine representing *office room*. ARC object may be coded to have more intelligence so that it can take decisions on behalf of the organizer, or it may be coded such that a notification window pops up at the remote machine when an object arrives. It then waits for user to perform actions on it. Various components of the application are described below.

7.2.1 Office Object Interface Specification

Following interface is used to create an ARC object in *workshop organization application*.

```
using System;
namespace OfficeObjectInterface{
   public interface IOffice{
      void setRoom(string room);
      string getRoom();
      void setDate(string date);
      string getDate();
      void setAvail(string status);
      string getAvail();
      void Permit();
      bool isPermitted();
   }
}
```

The ARC object created using above interface is used in modeling interaction with *office room*. Similar interface is used to create another ARC object to model interaction with *canteen room* in the example application. These interfaces are used to create *Generated Interfaces* of respective ARC objects.

7.2.2 Implementation of the Office Object Interface

Implementation of interface *IOffice* and interface *ITrigger* are done in class *Real*. *Trigger* method creates a proxy to the ARC object and registers the proxy. It also registers the proxy at the *NameServer*, which results in intimation to office application program. *Trigger* method terminates when office executive performs some events to permit the object to go back.

```
public class Real:PReal,OfficeObjectInterface.IOffice,ITrigger {
   ... private data member definitions ...
   public void Trigger(){
     // 1. create and register proxy
```

```
      RegisterForRemoteAccess();
      // 2. register with nameserver
      ARCObjectInterface_NamingService.IARCObject nameserver = null;
      if(this.Proxy != null){
          Object proxy = this.Proxy;
          nameserver = (ARCObjectInterface_NamingService.IARCObject)
                        Activator.GetObject(typeof(
                        ARCObjectInterface_NamingService.IARCObject),
                             "tcp://localhost:9123/NameServerClass");
          nameserver.Register("OfficeARCObject",proxy);
       }
       // 3. wait until permitted to go back
       while(!this.isPermitted()){
          Thread.Sleep(2000);
       }
       // 4. unregister proxy
       UnRegisterForRemoteAccess();
       // 5. unregister with nameserver
       if(nameserver != null)
           nameserver.UnRegister("OfficeARCObject");
   }

   ... implementation of other member functions ...
}
```

7.2.3 GUI of Workshop Organization Application

Figure 7.2 shows the GUI for *workshop organization application*. Organizer has to open and fill up both office form and canteen form. Filling the forms will enable send buttons. Clicking on the enabled send buttons pushes ARC objects to respective destination locations. Clicking on send buttons disables corresponding open buttons.

As shown in Section 7.2.2, office ARC object is coded such that it goes to the office and registers its proxy with the *NameServer*. Program representing office gets a notification from the *NameServer*. Office executive has to commit the reply to permit the office ARC object to go back to the sender. When object comes back to the originator, open button will be enabled. Clicking on open button shows office form including the changes made at remote machine.

7.2.4 Handling Transactions at Local Organizing Sites

Local organizing sites, like *office room* and *canteen room* in the current example application, run daemons to handle incoming ARC objects. These daemons get an intimation from *NameServer* on arrival of a new ARC object. Figure 7.3 shows a notification window,

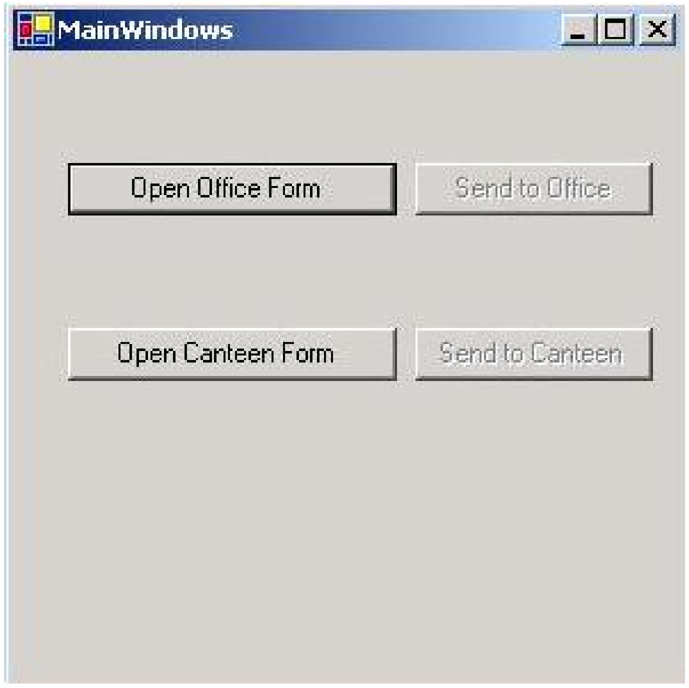


Figure 7.2: Workshop Organization Application GUI

which pops up on receipt of an intimation from *NameServer* about arrival of an ARC object . At *office* site, clicking on *OK* button of notification window shows office form as shown in Figure 7.4. Office executive may fill up the required data and click *Done* button to return the ARC object to the originator machine.

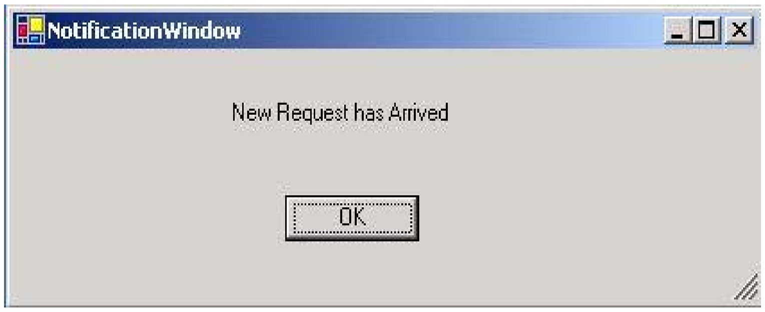
7.3 Disconnected Operation Using ARC over the Internet

This section illustrates use of Activation and Deactivation services over the Internet with an example. In the disconnected operation, an ARC object migrates to a machine and deactivates itself by writing its state to harddisk file. Subsequently machine may get disconnected and user applications at that machine may deserialize the object state to work with it. Finally an application may serialize the object to a file on harddisk. When machine is connected to the network, ARC object may be reactivated and sent out of the machine. Various components of the application are described below.

7.3.1 Interface Specification

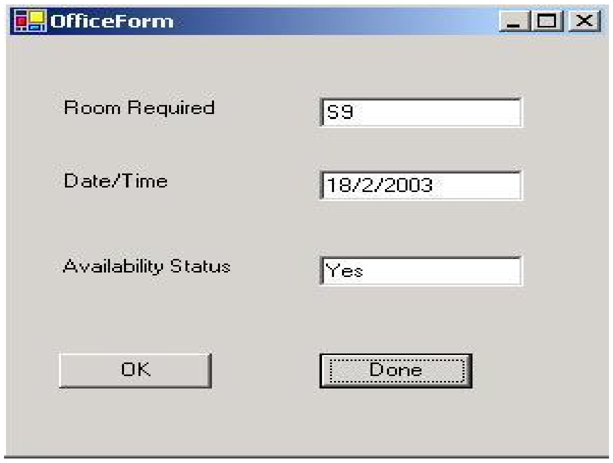
Following is the interface of an ARC object.

```
using System;
```


NotificationWindow

New Request has Arrived

Figure 7.3: Notification Window at Local Organizing Site


OfficeForm

Room Required	S9
Date/Time	18/2/2003
Availability Status	Yes

Figure 7.4: Office Form Used to Model Interaction with Office Room in the Application

```
namespace ARCobjectInterface{
    public interface Iarcobject{
       // member functions of ARC object
       void setStatus(int i);
       int getStatus();
    }
}
```

7.3.2 Implementation of the Interface

ARC object, after reaching a remote machine serializes itself to a file on the harddisk of remote machine. Applications at remote machines may deserialize the object and work with the object before reactivating. Following implementation of method *Trigger* shows that when ARC object is reactivated, the ARC object goes back to originator application due to termination of method *Trigger*.

```
[Serializable]
public class Real:PReal,ARCobjectInterface.Iarcobject,ITrigger{
   private bool flag = false;
   public void Trigger(){
      if(!flag){
        flag=true;
        SerializationNamespace.Serializer.
            SerializeAndDeactivate("ARCobject.tvk",this);
       }
   }
   public void setStatus(int i){
         // code that changes object status
   }
   public int getStatus(){
         // code that returns object status
   }
   ... other member functions ...
}
```

7.3.3 User Application at Remote machine

User applications at remote machine may deserialize the object state from the serialized file on the harddisk to work with it. Following code sample shows the usage of the library provided for Activation and Deactivation service described in Section 4.3.4.

```
public class Class1{
  public static void Main(){

   //Deserialize ARC object from a harddisk file
   ARCobjectInterface.Iarcobject obj = (ARCobjectInterface.Iarcobject)
             SerializationNamespace.Serializer.deserialize("filepath");

   //Work with the object locally

   //Serialize ARC object to a harddisk file
   SerializationNamespace.Serializer.Serialize("filepath",obj);

   //Deserialize and Activate the ARC object
   SerializationNamespace.Serializer.deSerializeAndActivate("filepath");
  }
}
```

Chapter 8

Conclusion and Future Work

Goal of the project was to design and implement an object-oriented ARC framework over Microsoft .NET platform. The report discussed architectural issues, deployment techniques and, design and implementation techniques for features of ARC framework software over the .NET. UML diagrams are extensively used to present these issues of ARC framework. The report also presented application development process over ARC framework.

Emphasis of the work was on applying a software development process for the framework. In the process, it was observed that building an architectural view of the framework simplified the software development process. Layered architecture for the LAN version of ARC software simplified extending ARC over the Internet by adding an extra layer to address differences between LAN and the Internet environments. It allowed software component reuse and also facilitated maintaining same abstraction of ARC framework for both LAN and the Internet versions.

ARC software is written on .NET Beta 2 version using C# programming language. Both LAN and the Internet versions of ARC were tested on four Windows 2000 workstations. Table 8.1 summarizes the project work by identifying various ARC services, their purpose, status of implementation of these services in both LAN and the Internet versions, and how these services are made available.

Some guidelines for future work from the implementation point of view are identified below.

- Security issues are not addressed in current work and needs to be addressed.
- Data structure for ring configuration in the Internet version can be modified to address geographic location advantage.
- *getHPFVector* for the Internet version has not been implemented, currently, *getHPF-Value* is used to acquire a lock on a single machine.
- Leave operation for both LAN and the Internet versions has to be developed. At present, killing the *Join* process is used for leaving.

- Applications presented in the report demonstrates the working of the ARC framework. Applications that involve some real world problem need to be found and solved using ARC features.

Services	Purpose of the Service	LAN version	Internet version	Provided as Daemon/Library (D/L)
ARC object Registration	Gets an object ID	yes	yes	L
getHPFVector	Anonymity in node selection	yes	no	D
getHPFValue	Explicit node selection	yes	yes	D
Lock in HPF value	Migration Assurance	yes	yes	An attribute in D
Release	Release acquired lock	yes	no	D
push	ARC object Migration	yes	yes	L
Trigger	A method of ARC object that is to be executed after object migration	yes	yes	L
Parallelism	Non-blocking nature of push operation	yes	yes	L
Auto Retraction	Asynchronous return of migrated ARC object	yes	yes	L
sync	Wait till retraction	yes	yes	L
Roaming	Multiple hopping of an ARC object	yes	yes	L
FTS Service register	Specify fault tolerance behavior	yes	yes	D
Auto-Execution	Guaranteed execution of trigger method in case of failures	yes	yes	L
ARC object registration with *NameServer*	Export interface of the ARC object to remote context after migration	yes	yes	L
User program registration with *NameServer*	Register a request for notification of an ARC object arrival	yes	yes	D
connect	Enables message invocation on migrated ARC object from originator application	yes	yes	L

Services	Purpose of the Service	LAN version	Internet version	Provided as Daemon/Library (D/L)
Join	Dynamic join of a node	yes	yes	D
Leave	Dynamic leave of a node	yes	no	D
Activation	Deserialize and activate an ARC object from harddisk file	yes	yes	L
Deactivation	Deserialze an ARC object to harddisk file	yes	yes	L

Table 8.1: Summary of the project

Appendix A

ARC Package Installation

What do you need?

ARC Software is available as a zip file. Unzipping the file creates various directories. Directory *compile* contains *makefile*. Running command *nmake* in that directory from a console creates executable *Join.exe* and all dll files within that directory. Current implementation is tested for execution on .NET Beta 2 platform.

Important files in the compile directory:

- join.exe
- configARC
- hosts
- And some dlls, which are discussed in Appendix B

Making changes in configARC file and hosts file:

configARC file:
ConfigARC file should contain three entries each separated by a new line character.

1. port number (eg. 9123)
2. home directory, i.e. path of the directory containing all the downloaded files. (eg. D:\\physical\\path_to\\downloaded\\directory\\)
3. Number of locks that the joining machine can give away to the requesting remote machines. (eg. 7)

Hosts file:
Hosts file contains IP addresses of all machines that could possibly join into ARC distributed network. Only, machines listed in this file can be used. A line in the file should contain only one IP address.

Note:

1. Port number should be equal at all machines whose IPs are listed in *hosts* file.
2. *hosts* file should be same every where.
3. Number of locks and directory location can be different and is left to users choice.
4. Notice that while specifying home directory in configARC file we have used two back-slashes. This is necessary for using it directly inside c# program.

Starting ARC distributed system:

For joining a machine into ARC network execute join.exe file contained in the *compile* directory.

Note:
Currently, to take the machine out of ARC network, kill the join.exe process that was started earlier. Graceful shutdown is yet to be tested.

Appendix B

Steps in Writing Hello World Application using ARC

Writing distributed application:

Writing application over ARC framework involves two parts, A and B. Part A is to make an ARC object and Part B is to develop distributed application using the ARC object made in Part A.

Part A:

This section assumes that the reader knows, what is namespace and related things. It is also assumed that the reader has used Microsoft Visual Studio before and knows things like creating a dll file and adding references to a project opened in Visual Studio. Though these are not essential to understand the process but for developing the application its required.

Making an ARC object involves 3 steps. Below we see them one by one.
Step 1:
Interface specification:

Open a new project and write an interface of the ARC object. Build the project to create the dll.

Example:

Below we see a sample interface. HelloInterface is the name given to the project. When we build the project we get helloInterface.dll inside \bin\debug directory of the project directory.

Code sample:

```
using System;
```

```
namespace HelloInterface{
  public interface IHello{
    void sayHello();
  }
}
```

Step 2:
Creating Generated Interfaces Namespace:

In this step, developer needs to hand code some of the required interfaces mentioned in the inheritance hierarchy. The inheritance hierarchy can be found in class diagrams shown in Figure 3.1 and Figure 3.2.

Example:

- Create a new project.
- Name it as HelloGeneratedInterface
- delete Class1.cs
- add IContainer and IMigratable classes to the project.
- add references to following dll files. Browse into the directory you have downloaded for these dll files.
  1. HelloInterface
  2. PushCommandInterface
  3. PushRequestInterface
  4. SyncInterface
  5. UserInterface_HPFValue

Note:
HelloInterface.dll is the dll created in step 1 above. Next sub step is to write IContainer and IMigratable interfaces. Following is the code sample of GeneratedInterfaces. It is easy to verify the inheritance relations by looking at class diagrams.
Code sample:
HelloGeneratedInterfaces.IContainer

```
using System;
using HelloInterface;
using SyncInterface;
namespace HelloGeneratedInterfaces{
   public interface IContainer :
             HelloInterface.IHello,SyncInterface.ISync {}
}
```

HelloGeneratedInterface.IMigratable

```
using System;
using PushRequestInterface;
using HelloInterface;
namespace HelloGeneratedInterfaces{
  public interface IMigratable:
            PushRequestInterface.IPushRequest,HelloInterface.IHello{}
}
```

Build the project to get HelloGenerateInterfaces.dll.

Step 3:
main namespace generation:

In this step the developer needs to hand code some default classes as well as needs to implement the interface given in step 1.

Example:

- create new project. Name it as HelloNamespace.
- Add Container.cs, Real.cs, FrontEndReal.cs, and Factory.cs class files to the project.
- Add references to following dll files
  1. ARCObjectInterface_ARCSystem.dll
  2. ARCObjectInterface_ObjectServer.dll
  3. ARCSystemInterface_ARCObject.dll
  4. CodeMotion.dll
  5. CodeMotionServer.dll
  6. FTSServiceInterface_ARCObject.dll
  7. HelloGeneratedInterfaces.dll
  8. HelloInterface.dll
  9. MessageDefinition.dll
  10. NextHopInterface.dll
  11. ObjectInfoInterface.dll
  12. ObjectIdInterface.dll
  13. PartialClassNamespace.dll
  14. PortMapper.dll
  15. PushCommandInterface.dll

16. PushRequestInterface.dll
17. RealObjectInterface_Proxy.dll
18. RealUtilInterface.dll
19. RetractionInterface.dll
20. SyncInterface.dll
21. TriggerInterface.dll
22. UserInterface_HPFValue.dll

All these dll files can be located inside the *compile* directory except dlls starting with Hello, since these are developed in previous steps. Container and FrontEndReal classes also, contain implementation of methods present in the interface specified in step 1. In both classes method body is typically delegation of function call to actual real object. Inside Factory.cs file the developer needs to mention the names of dll files to be transferred to remote machine. These files contain compiled code of ARC object. In our example these files are

1. HelloInterface.dll
2. HelloGeneratedInterface.dll
3. PartialClassNamespace.dll
4. HelloNamespace.dll

Inside RealObj.cs, the developer needs to implement Trigger method and sayHello method. Building the project creates helloNamespace.dll file in the project's bin\debug directory.

This completes creation of ARC object. It could be noticed that the development of ARC object started with specifying an interface of ARC object and ended with implementing that interface plus Trigger method. In-between steps are supposed to be generated by preprocessor when given an ARC object interface as an input. Current development process requires hand coding all these files and hence needs special care in keeping proper names at required places.

Part B:

Generated ARC object may be used in writing the distributed application. This part shows development of a sample distributed application, which sends an ARC object to a remote machine to print *Hello* message at that remote machine.

Example:

- Create new project and name it as hellotest

- Refer all dlls present in HelloNamespaces bin\debug directory.
- Add reference to UserInterface_HPFServer.dll
- Code the application logic.
- Build the project to create hellotest.exe file in its bin\debug directory.
- Copy following files to hellotests bin\debug directory before executing the application.
    1. HPFServer.dll
    2. PeerInterface_HPFServer.dll
    3. ARCSystemInterface_HPFValue.dll
    4. ARCSystemInterface_HPFVector.dll
    5. ARCSystemInterface_HPFServer.dll
    6. LeaveInterface_HPFServer.dll

Execute the program.
Note:

Point to be noted here is that above mentioned files are dependencies of HPFServer, and because it is used from hellotest project, these files needs to be copied. Similarly if FTSService or NameService were to be used then corresponding server's dependencies should also be present to avoid runtime exceptions.

Code Sample:

```
using System;
using System.Threading;
using HelloGeneratedInterfaces;
using HelloNamespace;
using UserInterface_HPFServer;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;

namespace hellotest{
  public class Class1{
    public static void Main(){
        TcpChannel chan = new TcpChannel(8106);
        ChannelServices.RegisterChannel(chan);
        UserInterface_HPFServer.IUser hpfserver =
             (UserInterface_HPFServer.IUser)Activator.GetObject
                  (typeof(UserInterface_HPFServer.IUser),
                  "tcp://localhost:9123/HPFServerClass");
        UserInterface_HPFValue.IUser hpf1=null;
        hpf1 = hpfserver.getHPFValue("10.105.24.13");
        HelloGeneratedInterfaces.IContainer arcobject = Factory.New();
        arcobject.push(mach13);
        arcobject.sync();
        Console.WriteLine("press enter to exit");
        Console.ReadLine();
    }
  }
}
```

References

[1] Jack Dongarra, Weicheng Jiang, Robert Manchek, Vaidy Sunderam, Al Geist, Adam Beguelin. PVM 3 Users guide and reference manual. 1994.

[2] Tom Archer. *Inside C#.* Microsoft Press, September 2001.

[3] L. Aruna. Support for dynamic task distribution on an RPC based ARC kernel. Master's thesis, Indian Institute of Technology Bombay, Jan 2001.

[4] Silberschatz, Galvin. *Operating System Concepts.* Addison Wesley, fifth edition, 1999.

[5] James Rumbaugh, Ivar Jacobson, Grady Booch. *The Unified Modeling Language Reference Manual.* Addison Wesley, 1999.

[6] Object Management Group. Common Object Request Broker Architecture Specification. *http://www.omg.org/*, 2002.

[7] Danny B. Lange and Mitsuru Oshima. Mobile Agents with Java: The Aglet API. *World Wide Web Journal*, 1998.

[8] David S. Platt. *Introducing Microsoft .NET.* Microsoft Press, January 2003.

[9] Dale Rogerson. *Inside COM.* Microsoft Press, 1997.

[10] Aruna L., Yamini Sharma, Rushikesh K. Joshi. Object-centric Design of an ARC Kernel. In *Proceedings of HPCN*, volume 2110, pages 251–262, 2001.

[11] D. Janaki Ram, Rushikesh K Joshi. Anonymous Remote Computing, A paradigm for Parallel Programming on interconnected Workstations. *IEEE Transactions on Software Engineering*, pages 75–90, Jan/Feb 1999.

[12] Thomas Wheeler. Reducing Development Effort using the Voyager ORB. *http://www.recursionsw.com*, 2003.