Javanaise For Distributed Java Objects

By
S.GUNASEKARAN.
Reg.No: 71203405004

of

KUMARAGURU COLLEGE OF TECHNOLOGY
COIMBATORE-641006.

A PROJECT REPORT
Submitted to the
FACULTY OF INFORMATION AND COMMUNICATION ENGINEERING

In partial fulfillment of the requirements
for the award of the Degree

of

MASTER OF ENGINEERING

IN

COMPUTER SCIENCE AND ENGINEERING

June,2005

BONAFIDE CERTIFICATE

Certified that this project report titled **"JAVANAISE FOR DISTRIBUTED JAVA OBJECTS"** is the bonafide work of **Mr.GUNASEKARAN.S**, who carried out the research under my supervision. Certified further, that to the best of my knowledge the work reported herein does not form part of any other project report of dissertation on the basis of which a degree or award was conferred on an earlier occasion on this or any other candidate.

GUIDE
[V.S AKSHAYA]

HEAD OF THE DEPARTMENT

The candidate with University Register No. **71203405004** was examined by us in the project viva-voce exam held on 22/6/05.

INTERNAL EXAMINER

EXTERNAL EXAMINER
22/6/05

ABSTRACT

Many distributed programming environments have been designed to support distributed shared objects over the Internet. Most of these environments (for example, Java RMI and CORBA) support client server applications where distributed objects reside on servers, which execute all methods (remote or local) invoked on the objects.

Traditional client-server models do not support client-side object caching and the local access of those objects. I believe that object caching is critical to distributed applications, especially over the Internet, where latency and bandwidth are highly variable. I have developed a configurable and efficient remote method invocation mechanism that provides the same interface as Java–RMI, while extending its functionality so that shared objects can be cached on the accessing nodes. The mechanism, called Javanaise, is based on the caching of clusters, which are groups of interdependent Java objects.

I have implemented a prototype of this service. The service consists of a set of system classes and a proxy generator implemented in the Java environment. The service and the applications that use it are dynamically deployed to client nodes at run-time, thanks to Java mobile code. The objects managed by the applications are transparently shared between the client nodes, so that the application developer can program as in a centralized setting.

This system support relies on object replication on the client nodes. Logically related objects are grouped in clusters, the cluster being the unit of sharing, replication and coherence. One of the main advantages of this proposal is that the object clustering policy is tightly coupled with the application code, thus ensuring locality, while keeping clustering transparent to the application programmer. This service had been prototyped and validated with simple distributed applications.

கருத்துச்சுருக்கம்

ஒருங்கமைக்கப்பட்ட பகிர்பயன்பாட்டு அமைப்பை இணையத்தில் இணைக்கும் போது இன்னல்கள் பல. சாதாரண தொலைதூர அமைப்பு முறையின் (RMI) கீழ் பெறுனன் கணினி (client) தனது தேவைக்கான சேவைக்கு அதன் காரணிகளை மட்டுமே வழங்கான் கணினிக்கு அனுப்பும், பின் வழங்கான் கணினி (server) அதை செயல்முறைபடுத்தி அதன் பதிளீட்டு விடையை பெறுனன் கணினிக்கு அனுப்பும்.

இந்த அமைப்பு முறையை அப்படியே இணையத்தில் உள்ள பகிர் பயன்பாட்டு முறையில் செயல்படுத்தினால், இணையத்தின் திறன், வேகம், செயல்பாடு குன்றும். ஏனெனில் இணையம் என்பது பலதரப்பட்ட வேகதிறன் கொண்ட வலையமப்புகளின் ஒருங்கினைப்பு ஆகும்.

இந்த குறையை குன்றச்செய்ய எனது செயல்முறைதிட்டமான ஜாவாநெய்ஸ் உதவுகிறது. இதில் ஜாவா வகுப்புப்பொருட்கள் (class objects) பயன்பாட்டின் அடிப்படையில் தொகுப்புகளாக (Clusters) தொகுக்கப்படுகிறது . இந்த தொகுப்புகள் பயன்பாட்டு வகை தொகுப்பு (Application Dependent Cluster) ஆகும். இது வழங்கான் கணினியில் அமுலாக்கப்படுகிறது.

இந்த பயன்பாட்டு வகை தொகுப்புகளில் ஏதேனும் ஒரு பொருளை பெறுனன் கணினி பயன்படுத்தும் பொழுது,அந்த தொகுப்பே பெறுனன் கணினிக்கு அனுப்பப்படுகிறது. எனவே பெறுனன் கணினி தொகுப்பில் மீதமுள்ள பொருளை செயல்படுத்த வழங்கான் கணினியை அனுகவேண்டிய அவசியமில்லை.

இதனால் இணையத்தின் வலையமைப்பை பயன்படுத்துதல் குறைக்கப்படும். எனவே இணையத்தின் திறன், வேகம், செயல்பாடு அதிகரிக்கும், மேலும் வழங்கான் கணினியின் பணியும் குறையும்.

ACKNOWLEDGEMENTS

Any endeavor over a long period of time can be successful only with the immense help, advice and hold up of many well-wisher, I take this opportunity to convey my sincere gratitude to all of them.

I express my profound gratitude to our esteemed principal **Dr.K.K.Padmanabhan, Ph.D.**, for his immense support throughout the project.

I sincerely express my gratitude to our Head of the department of Computer Science and Engineering **Prof.S.Thangasamy, Ph.D.**, for his valuable guidance and untiring help when doing my project work.

I would like to express my heart felt thanks to my guide **Ms.V.S.Akshaya,M.E.** Lecturer of Computer Science and Engineering Department for her valuable guidance and untiring help throughout the project.

I express my sincere thanks to our course coordinator **Mrs.L.S.Jayashree M.E.,** Senior Lecturer of Computer Science and Engineering department for her motivation.

I record my sincere thanks to our project coordinator **Mr.R.Dinesh M.S.,** Assistant Professor of Computer Science and Engineering Department for his constant encouragement and motivation

TABLE OF CONTENT

LIST OF TABLE

LIST OF FIGURES

CHAPTER-1

INTRODUCTION

Support for cooperative distributed applications is an important direction of computer systems research, involving developments in operating systems as well as in programming languages. More recently, the growth of the Internet, which is now daily used as a cooperation support, logically leads to consider the deployment of distributed cooperative applications over the Internet. Today, distributing applications on the Internet is closely linked with the Web (essentially URLs) and Java. Therefore, a first attempt to provide distributed shared objects on the Internet was Java-RMI, which provides remote method invocation between Java objects. Shared objects are uniquely named with URLs and a mechanism called *object serialization* allows distributed programs to exchange copies of objects (as in Sun RPC [rpcgen88]). However, using the RMI facilities, distributed applications are based on the client-server architecture, which does not allow objects to be cached and therefore accessed locally. It is possible to manage object replicas using the object serialization facility, but the coherence between the replicas has to be explicitly managed by the application programmer. I believe that object caching is one of the key features required by cooperative applications, especially over the Internet, whose latency and bandwidth are highly variable.

In order to assist the programmer, I propose a new system service, which implements the abstraction of a distributed shared Java object space. Objects are brought on demand on the requesting nodes and are cached until invalidated by the coherence protocol. With this system support, the programmer can

develop its application as if it were to be executed in a centralized configuration. Then, the application can be configured for a distributed setting without any modification to the application source code. Annotating the interfaces of the objects that are distributed, specifying the synchronization and consistency protocols to apply to these objects, performs this configuration. A prototype of this service has been implemented using java architecture and consists a proxy - generator which is used to generate indirection objects (proxies) for the support of dynamic binding, and a few system classes that implement consistency protocols and synchronization functions.

The main advantages of this approach are:

- Dynamic deployment. Applications are dynamically deployed to the client nodes from the node that hosts the application; thus we don't require applications to be statically installed prior to execution.
- Transparency. A distributed cooperative application can be developed as if it were to be run centralized. Distribution and synchronization are programmed separately from the application code.
- Caching the system support allows shared Java objects to be cached on cooperating nodes, thus enabling local invocation on distributed objects and reducing latency.
- Clustering. Grouping objects in clusters is one of the key techniques for achieving good performance by factorizing system costs.

CHAPTER-2

MOTIVATION

The main motivation for Javanaise is to provide adequate support for developing and executing cooperative applications on the Internet. Cooperative applications aim at assisting the cooperation between a set of users involved in a common task. An example of cooperative application is a structured editor which allows documents to be shared concurrently by remote users.

These applications are characterized by a large amount of shared data structures which are browsed or edited by cooperating users connected from remote workstations. Since these data structures should be brought to the accessing nodes, at least to be displayed and sometimes to be modified, a caching strategy should be used. Defining the unit of sharing and consistency is the key issue to efficiency.

Another important issue for this service is to facilitate the installation and administration of software. In an intranet, application installation is often managed by system administrators who are responsible for installing these applications properly and also ensuring that they do not act as Trojan horses in the intranet. However, requiring any cooperative application to be officially installed (by administrators) is constraining; in addition, it is a difficult issue for administrators to decide that an application can be trusted. An alternative is to

allow these cooperative applications to be freely downloaded on a Java virtual machine (just like applets), thus benefiting from a dynamic deployment of the applications with the guarantee that the application cannot corrupt the local host, thanks to Java's type safety.

Finally, I want to allow programmers to develop application as if they were to be run centralized. Then, a programmer can debug and test its application on a single machine, and then after a simple configuration step, run it in a distributed environment using this system support.

The three motivations described above (efficiency, easy administration, easy development) constitute the guideline, which leads us to the design of Javanaise.

CHAPTER-3

LITERATURE SURVEY

3.1) Introduction to RMI.

☑ Sun Microsystems, "Java Remote Method Invocation (RMI)," homepage, http://java.sun.com/products/jdk/rmi/.

RMI applications are often comprised of two separate programs: a server and a client. A typical server application creates some remote objects, makes references to them accessible, and waits for clients to invoke methods on these remote objects.

A typical client application gets a remote reference to one or more remote objects in the server and then invokes methods on them. RMI provides the mechanism by which the server and the client communicate and pass information back and forth. Such an application is sometimes referred to as a distributed *object application.*

Distributed object applications need to

☑ *Locate remote objects*:

Applications can use one of two mechanisms to obtain references to remote objects. An application can register its remote objects with RMI' s simple naming facility, the rmi registry, or the application can pass and return remote object references as part of its normal operation.

☑ *Communicate with remote objects*:

Details of communication between remote objects are handled by RMI; to the programmer, remote communication looks like a standard Java method invocation.

☑ *Load class byte codes for objects that are passed around*:

RMI allows a caller to pass objects to remote objects, RMI provides the necessary mechanisms for loading an object's code, as well as for transmitting its data.

The following illustration depicts an RMI distributed application that uses the registry to obtain a reference to a remote object. The server calls the registry to associate (or bind) a name with a remote object. The client looks up the remote object by its name in the server's registry and then invokes a method on it. The illustration also shows that the RMI system uses an existing Web server to load class byte codes, from server to client and from client to server, for objects when needed.

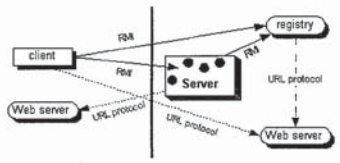

Figure-3.1 RMI Concept

Creating Distributed Applications Using RMI

- Design and implement the components of distributed application.
- Compile sources and generate stubs.
- Make classes network accessible.
- Start the application.

3.2) Distributed Object Model for the Java System

☑ Wollrath, R. Riggs, J. Waldo, "A Distributed Object Model for the Java System", *Computing Systems*.vol 9,no 4,2000.

Distributed systems require entities, which reside in different address spaces, potentially on different machines, to communicate. The Java system provides a basic communication mechanism, sockets . While flexible and sufficient for general communication, the use of sockets requires the client and server using this medium to engage in some application-level protocol to encode and decode messages for exchange. Design of such protocols is cumbersome and can be error-prone.

An alternative to sockets is Remote Procedure Call . RPC systems abstract the communication interface to the level of a procedure call. In order to support distributed objects in Java, we have designed a remote method invocation system that is specifically tailored to operate in the Java environment. The Java language's RMI system assumes the homogeneous environment of the Java Virtual Machine, and the system can therefore follow the Java object model whenever possible.

several important goals for supporting distributed objects in Java:

- support seamless remote invocation between Java objects in different virtual machines;
- Integrate the distributed object model into the Java language in a natural way while retaining most of Java's object semantics;
- make differences between the distributed object model and the local Java object model apparent;
- minimize complexity seen by the clients that use remote objects and the servers that implement them;
- preserve the safety provided by the Java runtime environment.

Parameter Passing in Remote Invocation

A parameter of any Java type can be passed in a remote all. These types include both Java primitive types and Java objects (both remote and non-remote). The parameter passing semantics for remote calls are the same as the Java semantics *except*:

- non-remote objects contained in a parameter of a remote call are passed by *copy*; and,
- non-remote objects returned as the result of a remote call are also passed by *copy*.

That is, when a non-remote object is passed in a remote call, the content of the non-remote object is copied before invoking the call on the remote object. Thus, there is no relationship between the non-remote object the client holds and the one it sends to a remote server in a call. For example, let's suppose that the remote object bank has a method to obtain the bank account given a name and social security number; the account information info is not a remote object but a local Java object:

Locating Remote Objects

A simple bootstrap name server is provided for storing named references to remote objects. A remote object reference can be stored using the URL-based interface java.rmi.Naming. For a client to invoke a method on a remote object, that client must first obtain a reference to the object. A reference to a remote object is usually obtained as a return value in a method call. The RMI system provides a simple bootstrap name server from which to obtain remote objects on given hosts. The Naming interface provides Uniform Resource Locator (URL) based methods to lookup, bind, rebind, unbind and list
the name and object pairings maintained on a particular host and port.

Architectural Overview

The three layers of the RMI system consist of the following:

- stub/skeletons — client-side stubs (proxies) and server-side skeletons
- remote reference layer — invocation behavior and reference semantics (e.g., unicast, multicast)
- Transport:-connection setup and management and remote object tracking

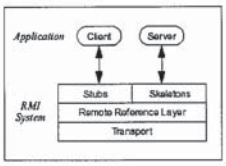

Figure-3.2 Architectural Overview

3.3) Distributed shared objects for Internet cooperative applications

☑ D.Hagimont, D. Louvegnies, "distributed shared objects for Internet cooperative applications", SIRAC Project INRIA, 655 av. de l'Europe, 38330 Mont bonnot Saint-Martin, France

Today, distributing applications on the Internet is closely linked with the Web (essentially URLs) and Java. Therefore, a first attempt to provide distributed shared objects on the Internet was Java-RMI which provides remote method invocation between Java objects. Shared objects are uniquely named with URLs and a mechanism called *object serialization* allows distributed programs to exchange copies of objects (as in Sun RPC). However, using the RMI facilities, distributed applications are based on the client-server architecture which does not allow objects to be cached and therefore accessed locally. It is possible to manage object replicas using the object serialization facility, but the coherence between the replicas has to be explicitly managed by the application programmer.

The main advantages of this approach are:

o Dynamic deployment :

Applications are dynamically deployed to the client nodes from the node that hosts the application; thus we don't require applications to be statically installed prior to execution.

o Transparency :

A distributed cooperative application can be developed as if it were to be run centralized. Distribution and synchronization are programmed separately from the application code.

o Caching :

This system support allows shared Java objects to be cached on cooperating nodes, thus enabling local invocation on distributed objects and reducing latency.

o Clustering:

Grouping objects in clusters is one of the key techniques for achieving good performance by factorizing system costs. Applications manage object groups in their internal structure and that clustering should be mapped on this application grouping.

3.4) Improving performances for non-functional properties

☑ D. Hagimont, N. De Palma, F. Boyer, S. Ben Athallat, "Improving performances for non-functional properties", Sardes project, F-38334 Saint Ismier, France,2002.

In a component-based middleware, an interaction between two components generally goes through two indirection objects, the first one being the exit port of the invoking component and the second the entry port of the invoked component. In the rest of the paper, we will refer to these indirection

- *Stub*: the stub object (*Stub_o2*) is used to manage dynamic binding of references to objects that may be brought dynamically from remote nodes. When the Java reference in the stub object is null (*ref_o2*), a copy of the referenced shared object is fetched, either locally if the object is already cached or remotely from a Javanaise server by using a unique identifier associated with the object (*id_o2*). There is one stub object per reference pointing to object *o2*.

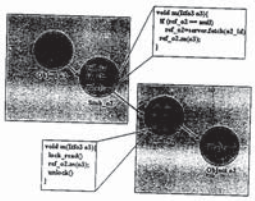

Figure-3.4 implementation based on indirection objects

- *Skeleton*: The skeleton object (*Skel_o2*) is used to manage invalidates and updates of the shared object (*o2*) according to a consistency protocol. It may fetch a new copy of O2 from the server and update the Java reference (*ref_o2*).

A stub is copied with the object, which includes its reference. The reference *ref_o2* in the stub is set to null when the stub is copied on a machine (it is a transient Java reference). The skeleton is copied with the referenced object (*o2*) the first time it is loaded on a machine. The reference *ref_o2* in the skeleton may be null subsequently to invalidations by the consistency protocol. Invalidation in the skeleton avoids having to invalidate all the stubs, which may reference the object.

3.5) Clustering technology

☑ BEA system,"About Clustering technology in weblogic",home page,www.e-doc.bea.com,©2001 BEA System inc, last updated 08/23/2000.

Clustering architecture

There are three general categories of traditional clustering architecture based on how each server in the cluster accesses memory and disks and whether servers share a copy of the operating system and the I/O subsystem. These three categories are:

1) Shared-memory

   All servers in the cluster use the same primary memory.

2) Shared –disk

   Each server has its own memory but the cluster shares common disks. Since every server can concurrently accesses every disk.

3) Shared –nothing

   Every server has its own memory and its own disk..

• Replication and clustering :

Replication which mirrors data and state from one server to another,is essential for all clustering architectures.Replication facilitates both scalability and availability .

• Persistence and clustering:

One alternatives to keeping in memory replicas of objects with internal state, particularly user data, is to keep stateful components of a distributed system in underlying persistence data store , preferably a transaction packed database.e

CHAPTER- 4

RELATED WORK

Wollrath proposed a remote method invocation facility between Java objects called Java RMI. It supports distributed shared objects on the Internet but does not allow objects to be cached and therefore accessed locally. It is possible to manage object replicas using the object serialization facility in Java RMI, but the coherence between the replicas has to be explicitly managed by the application programmer.

Some projects have addressed the problem of object caching, but the proposed solutions are not targeted to the Java environment .The Hybrid Adaptive Cache and Thor projects at MIT address the problem of managing client caches in distributed and persistent object storage systems.

The objective is to provide hybrid and adaptive caching, which manages both page caches and object caches, according to an object's behavior. Other environments, such as the Object Store database management system, use object caching to meet scalability requirements. Javanaise is distinguished from these environments by providing an RMI–based solution that relies on cluster caching for general-purpose applications.

Implementation of Java RMI, extending it to benefit from both object caching and the UDP communication protocol (which is faster than TCP).We

share the same objectives regarding object caching ; however, their extension appears to be deeply integrated within the JDK1.1.5.Thus, their solutions assume the widespread use of this modified environment. Javanaise, on the other hand, is built entirely on a standard Java environment; all its components can be dynamically loaded with application code. Chockler propose a scalable caching service for Corba objects, based on a hierarchical cache architecture.

This service uses domain caching servers to cache objects as close as possible to clients. This contrasts with Javanaise, which allows an object to be cached on the client side. Some other CORBA platforms provide adaptation features, but do not yet directly address the problem of object caching. The OpenCorba project aims at providing an adaptable object broker that can reify an object's internal mechanisms and then adapt them at runtime. Allowing object behavior to be dynamically modified at runtime in turn allows different strategies to be used for dynamic placement of objects. Open Corba can thus be considered as a useful support for the provision of an adaptable RMI.

CHAPTER-5

LINE OF ATTACK

☑ **Cluster Designing .**

Clusters are inter-dependent java objects. Javanaise uses Application Dependent clustering which manage logical graphs of objects in their data structures. A cluster is identified by a Java reference to a first object (called a *cluster object*) and the graph that defines the cluster is composed of all the Java objects that are accessible from the cluster object (the transitive closure also applicable).

The boundaries of this graph are defined by the leaves of the graph and by the references to other cluster objects. A reference to another cluster object is called an *inter-cluster reference*. The Java objects within a cluster are called *local objects*.

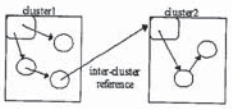

Figure-5.1) Inter Cluster Reference

A cluster object is an instance of a class which has been defined as being a cluster class. Only interfaces of cluster objects

CHAPTER-6

BASIC DESIGN CHOICES

6.1. Managing clusters

The main problem we have to solve is to efficiently manage distributed replicas of Java objects while keeping distribution transparent to the application programmer. Managing object replicas requires mechanisms for faulting on objects, invalidating and updating objects in order to ensure consistency. These mechanisms should be hidden to the application programmer, who should only manipulate Java references as if every object were local.

A cluster of objects is a group of objects which is supposed to be coarser grain (than a single object). Therefore, since system mechanisms are generally applied to coarse grained resources (e.g. IOs), they are applied to clusters, thus factorizing the costs of these mechanisms for all the objects within a cluster. However, clustering works well only if objects co-located within the same cluster are effectively closely related at execution time.

The mechanisms factorized here are naming, binding and consistency mechanisms. In order to be able to dynamically bind a reference to a remote object, we need to associate a unique name with each object, thus allowing the object to be located and brought to the requesting node. In order to implement

object binding [1], we need to manage indirection objects that allow object faults to be triggered if the reference is not yet bound. In order to manage objects consistency, we need to exchange messages between cooperating nodes to invalidate and update copies according to a consistency model.

Managing clusters of objects is a means for amortizing these costs (indirection objects, messages) over a group of objects that are inter-dependent. Inter-dependence means here that if one object of the group is accessed, most of the objects included in the group are likely to be used in the near future.

6.2. Application dependent clustering

The system exports to applications a cluster management interface allowing objects to be stored in or migrated to any cluster. From the programmer's point of view, managing clustering is complex and most of the time leads to a default policy, which is inefficient and doesn't actually use the flexibility of the clustering interface.

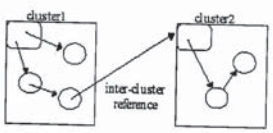

Figure 6.1). Management of clusters

[1] without modification to the Java virtual machine

In Javanaise, *application dependent clustering* had been implemented. This approach is inspired by the observation that cooperative applications tend to manage logical graphs of objects in their data structures. For example, a cooperative structured editor manages chapters that are composed of sections, themselves composed of subsections and paragraphs. I claim that some of these graphs should be managed as clusters by the system since they correspond to closely related objects according to the application semantics.

In Javanaise, a cluster is an application-defined graph of Java objects. A cluster is identified by a Java reference to a first object (called a *cluster object*) and the graph that defines the cluster is composed of all the Java objects that are accessible from the cluster object (the transitive closure). The boundaries of this graph are defined by the leaves of the graph and by the references to other cluster objects. A reference to another cluster object is called an *inter-cluster reference* (Figure 6.1). The Java objects within a cluster are called *local objects*.

A cluster object is an instance of a class (defined by the programmer) which has been defined (when the application is configured to be run distributed) as being a cluster class. Only interfaces of cluster objects are exported to other cluster objects, which means that the interface of a cluster object may only include methods whose reference parameters are references to cluster objects. Therefore, local objects in one cluster are only accessible from objects within the cluster.

6.3. Application programming

The programmer develops applications using the Java language without any language extension nor system support classes (libraries). An application can be debugged and tested locally (on one machine). Configuring the application for distribution first consists in specifying which classes are cluster

classes. The configurator should take into account the data structures managed in the application, i.e. the links between the classes that compose the application. However, this separation between the configuration and the application code makes it possible to experiment with different configurations for the same application without any modification to the application.

Since an application is developed centralized, it does not deals with synchronization and consistency problems. A second step in the configuration is to associate synchronization and a consistency protocol with each cluster. This is done at the level of the interfaces of the cluster classes. The interfaces of the cluster classes are annotated with keywords that define the consistency and synchronization protocols associated with the clusters.

At the moment, i have only implemented a single reader / multiple writers protocol. In the interface of a cluster, it is possible to associate a mode (reader or writer) with each method. When the method is invoked on a cluster instance, a lock in that mode is taken and a consistent copy of the cluster is brought to the local host. However, i will experiment with different consistency/synchronization protocols in the near future.

CHAPTER-7

IMPLEMENTATION PRINCIPLES

7.1. Managing cluster binding

Since a cluster is a graph of Java objects, clusters may be brought dynamically on a requesting node using the Java serialization mechanism. The problem is to manage dynamic binding of references to objects that may be brought dynamically from remote nodes. Since the unit of naming and caching is the cluster, I had provided a mechanism for dynamic binding of inter-cluster references. This implementation relies on intermediate objects called *proxies* that are transparently inserted between the referenced cluster and the cluster which contains the reference (Figure 7.1). A proxy contains a Java reference that points to the referenced cluster object if it is already there and null if not. It also contains a unique name associated with the cluster, allowing the cluster to be located and a copy to be brought on the local host.

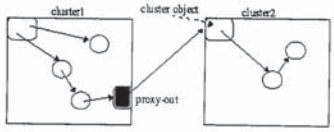

Figure 7.1. Binding of inter-cluster references

The class of the proxy object is generated from the interface of the cluster class to which it points. This proxy implements the same interface as the cluster object. Each method invocation is forwarded to the actual cluster object if the reference is already bound, i.e. if the Java reference in the proxy is not null. If this Java reference is null, then a function of the runtime system is invoked in order to check whether the cluster is already cached (subsequently to the binding of another inter-cluster reference). A copy of the cluster is fetched if required and the Java reference in the proxy object is updated. In the following, i call this proxy a *proxy-out object*. Proxy -out objects are stored in the cluster which contains the reference to the cluster.

7.2. Managing cluster consistency and synchronization

First, the problem is to manage invalidates and updates of clusters according to a consistency protocol. Here the mechanism is independent from the consistency protocol .

A cluster can be invalidated on one node (Java virtual machine) simply by assigning to null the Java references in the proxy -out objects that reference the cluster. All the Java objects included in the invalidated cluster are then automatically garbage collected by the Java runtime. However, instead of dynamically looking for all the proxy -out objects that point to the invalidated cluster (which would be complex and inefficient), i decided to manage another type of proxy called *proxy-in object*, which is inserted between the proxy -out object and the cluster it points to (Figure 7.2). A proxy -in object is stored in the cluster which is referenced. Similarly to proxy - out objects, a proxy-in object forwards method invocations to the referenced cluster if its internal Java

In Figure 7.3, a local object in *cluster1* (1) performs an invocation (*c3 = c2.meth()*) on *cluster2*. Invoked method returns a Java reference stored in *cluster2* (2), which is a reference to *cluster3* . In order able to store a reference to *cluster3* in *cluster1*, the system must create a proxy-out which points to *cluster3* This proxy -out object is created by the proxy -out in *cluster1* which is associated with the reference to *cluster2* (4). An onward reference parameter would be managed similarly by the proxy -in object in *cluster2*. When managing proxy -out objects for entering parameters in a cluster, it is needed to guarantee that all references to a cluster *C* within the cluster point to the same proxy-out object. This is especially important comparing two variables that contain cluster references (within one cluster). To do this, manage a table in the cluster which registers the proxy -out objects which already exist in the cluster. When a reference enters the cluster and if an associated proxy-out object already exists in the cluster, then this proxy -out object is and no additional proxy -out object is created. Therefore, having two proxy-out objects associated with the same external reference in one cluster is avoided.

7.4. Other clustering issues

In this section, consider two variants of clustering model which are clusters with several entry points and re-clustering. Both variants preserve the model in the sense that they don't require hard coding a clustering policy in the application code. The clustering policy remains based on the data structures and the interfaces managed in the application.

First consider the management of several entry points in clusters. I have imposed the restriction that any interaction with a cluster should take the form of a method invocation on the cluster object (the unique entry point of the cluster).

This implies that reference parameters in the interface of a cluster are always references to cluster objects. However, cluster interfaces could be allowed to include local object reference parameters. This would require the ability to dynamically install proxy -out and proxy -in objects, thus managing several entry points in clusters. It would not be difficult to extend my prototype in order to implement this variant of the model.

The second variant is to provide support for reclustering. Reclustering means here the ability to move a local object from one cluster to another. The interface of the cluster may include a "move" statement associated with the reference parameter, meaning that the local object should be moved to the destination cluster. Then the local object in the source cluster must be transformed into a proxy-out object, and the proxy -out object (if there is one associated with this local object) in the destination cluster must be changed into the migrated local object. These two variants constitute interesting perspectives since they preserve the spirit of this clustering model.

CHAPTER-8

PROTOTYPE ON JAVA

I have implemented a prototype of this system support and its description as follows.

8.1. Overall architecture

The overall architecture is illustrated on figure 8.1. In this prototype, assume that the code of Javanaise, the code of the application and the persistent clusters are stored on a node called the Home Site of the application. The code of Javanaise and the code of the application are dynamically deployed to the client nodes when the application is invoked.

An application is made available on the Home Site through a Web server and identified and located with a URL. The application is launched using an Applet viewer, the code of the application and of Javanaise being downloaded on client nodes just like an Applet. Clusters are then fetched on demand by the requesting nodes and shared following the consistency protocol.

The Javanaise system support consists of:

- The proxy generator which generates proxy-out and proxy -in classes from the interface of a cluster. Proxy -out and proxy -in classes provide mechanisms for reference binding and consistency management.

- some system primitives (*Javanaise client*) that are available on the client nodes and used by the proxies on these nodes. *Javanaise client* maintains a table of the clusters (proxy -in objects) that are already present on the local host.
- some system code (*Javanaise server*) that participates in the binding and consistency protocols on the home site. The Javanaise server maintains a table which registers the locations and locks held for all the clusters.

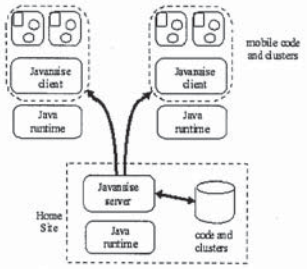

Figure 8.1 . Architecture

8.2. Generation of proxies

Proxies are generated from the interface of a cluster class using a proxy generator. Let's see the Java code skeleton of the proxy -out and proxy -in classes generated from the cluster interface below.

```
public class Cluster1 implements Cluster1_itf {
public void method1 (Cluster2_itf obj); : read
public Cluster3_itf method2 (); : write
```



```
}
```

This definition describes the interface of the cluster class *Cluster1* which implements the *Cluster1_itf* interface. Two methods are defined, the first one taking a onward reference parameter and the second returning a reference parameter. The first method requires a read lock before execution and the second a write lock. Below is the proxy -out class generated from the above interface definition. The name of this class is actually *Cluster1*, because when a program manipulate a reference to a *Cluster1* instance, it is in fact a reference to a proxy -out object which implements the same interface. This class defines two instance variables, *object_id* which is the unique identifier of the object the proxy - out refers to, and *proxy-in* which points to the proxy-in object. The identifier is unique in the context of the application; my prototype currently manages shared objects between applications located on a single Home Site, but it can be extended to manage different Home Sites simply by using URLs.

```
public class Cluster1 implements Cluster1_itf {
int object_id;
Proxy_in_Cluster1 proxy_in;
public Cluster1 () {
proxy_in = new Proxy_in_Cluster1();
object_id = proxy_in.my_id();
}
public void method1 (Cluster2 obj) {
if (proxy_in == null)
proxy_in = (Proxy_in_Cluster1) javanaise_client.get_proxy_in(object_id,
read);
proxy_in.method1(obj);
}
public Cluster3 method2 () {
```

```
if (proxy_in == null)
proxy_in = (Proxy_in_Cluster1) javanaise_client.get_proxy_in(object_id,
write);
Cluster1 p = proxy_in.method2();
p = clone_proxy-out(p); // clone proxy-out for backward parameter
return p;
}}
```

The first method is the constructor. When a user program invokes the creation of a *Cluster1* object, actually creates a proxy -out object which in turns creates a proxy-in object which creates the real instance of *Cluster1* (see below in the proxy -in class). If a constructor with parameters is defined, a constructor is generated accordingly in the proxy -out and proxy -in classes and implements the same interface. The two other methods check the binding of the reference to the cluster. If the reference has not already been bound, *Javanaise client* is invoked is order to get a copy of the cluster (including the proxy -in object) with lock in the corresponding mode[2]. Then the invocation is forwarded to the proxy-in object. Notice that the *method2* method returns a reference to a *Cluster3* instance.

The reference returned by this method is a reference to a proxy -out object in the invoked cluster. Then, i must create a clone of this proxy-out object, which will be stored in the cluster receiving the reference. This clone gets created only if a proxy-out object for the received reference doesn't yet exist in the receiving cluster. The table which registers the existing proxy -out objects in the cluster is stored in the cluster itself.

---

[2] The locking mode is specified here in order to get both the copy of the cluster and the lock on that cluster in a single request.

Below is the generated proxy -in class. The proxy -in class also defines a variable *object_id* for the unique identifier of the object and a variable *object* which points to the real cluster instance. It also contains a variable *lock* which indicates the lock that is held on the local host on this cluster (read, write or none).

```
public class Proxy-in_Cluster1 implements Cluster1_itf {
int object_id;
Real_Cluster1 object;
int lock;
int num_readers;
public Proxy_in_Cluster1 () {
object = new Real_Cluster1();
object_id = javanaise_client.register_id(this);
num_readers = 0;
}
public void method1 (Cluster2 obj) {
if ((object == null) || (lock == none)) {
object = (Real_Cluster1) javanaise_client.get_object(object_id, read);
lock = read;
}
Cluster2 p = clone_proxy_out(obj); // clone proxy-out for onward
parameter
num_readers ++;
object.method1(p); // effective call
num_readers --;
if ((lock == read) && (num_readers == 0)) {
javanaise_client.release_lock(object_id, read);
lock = none;
}
```



```
}
public Cluster3 method2 () {
if ((object == null) || (lock != write)) {
object = (Real_Cluster1) javanaise_client.get_object(object_id, write);
lock = write;
}
object.method1(obj); // effective call
javanaise_client.release_lock(object_id, write);
}
}
```

The constructor of the proxy-in class creates the real instance of *Cluster1* which has been renamed in *Real_Cluster1* (by the proxy generator) since the proxy-out class has been renamed in *Cluster1*. Then, *Javanaise client* is invoked in order to allocate a unique object identifier[3]. A reference to the proxy-in object is passed in order to initialize Javanaise internal tables (detailed below).

The two methods checks whether a copy of the cluster with the requested lock is present. If not, a copy and/or a lock are requested to *Javanaise client*. The locking policy currently implemented allows multiple readers and one writer at a time.

A lock held on a cluster is managed in the proxy -in object of the cluster. The proxy -in object invokes Javanaise client in order to request the lock. The proxy-in object also includes primitives (up calls not present in the above skeleton) that may be invoked by Javanaise client in order to check the status of

---

[3] In the current prototype, this allocation is forwarded to Javanaise server which ensure the identifier uniqueness.

the lock on the cluster and block a lock request from a remote node until the lock is explicitly released. The methods in the proxy –in object which manipulate the lock variable are synchronized.

8.3. Management of consistency and synchronization

In order to manage consistency and synchronization, two tables are maintained, one in *Javanaise client* and one in *Javanaise server*.

First, in *Javanaise server*, It is needed to locate any cluster and any lock. *ServerTable* is a table which keeps track of all the clusters that have an image in memory (see next section for clusters stored on disk). This table associates with each cluster (known by its *object_id*) the locations (node addresses) of all the images of the cluster in memory. If the cluster is in read mode, the table gives a list of the nodes that obtained a read lock on the cluster. If the cluster is in write mode, the table gives the address of the node which hosts the unique copy. This table is used in order to locate one copy (in read mode) or the last copy of the cluster. When a collision on a lock occurs (the cluster is locked in write mode), the request is sent to the writer node and *Javanaise client* responds only when the lock is released.

The second table (*Client Table*) is managed in *Javanaise client*. This table records the clusters that are cached on the local host. This table associates with each cluster (*object_id*) the Java reference to the proxy –in object which represents the cluster. Using the *Client Table*, *Javanaise client* can check the status of the cluster on this node and wait for a lock to be released by the proxy-in object (with the *release_lock* primitive). All the interactions between Javanaise clients and Javanaise server are based on message passing using the Socket interface. A message always includes a header object which describes the

message type. This header may be followed by (i.e. point to) a cluster object which is a graph of Java objects. These objects, the header and the cluster, are serialized in order to obtain a flat string of bytes which is sent on the socket. In order to be serializable, each object of the application must implement the *Serializable* Java interface, which means that it inherits default methods that are invoked to serialize the object. The default serialization behavior is to flatten the object state and to do it recursively following every Java reference in the object state. To stop this recursion, I redefined these serialization methods for the proxy -out objects: I only save the *object_id* field of the object and reset to null the Java reference when the object is deserialized.

8.4. Management of persistence

In Javanaise, clusters are persistent, which means they survive the application that created them. Clusters are stored on disk on the Home Site of the application, one file per cluster. Recall that each cluster is identified with an *object_id* (an integer) which is a unique identifier. In Javanaise server, the *ServerTable* keeps track of all the clusters that have an image in memory on one of the client hosts. When a cluster is requested and the cluster is not represented in memory, the cluster is read from its storage file. The byte stream read from the file is de-serialized and sent to the requesting client (and the *ServerTable* updated). When an application terminates on a client node and if some modified clusters are no longer used (cached on any node), they are serialized and copied back to their storage files on the Home Site.

8.5. Deployment of an application

Applications in Javanaise are deployed dynamically to the client nodes just like Applets. A Javanaise application is made available as an Applet through

a Web server and a client typically starts the application using an Applet viewer. This Applet contains the *main* entry point of the program. An initialization primitive is provided, which initializes the Javanaise environment, but also returns a Java reference to a name server object. This name server allows symbolic names to be associated with Cluster objects references. The *register* method registers a cluster reference with a given symbolic name and the *lookup* method returns the cluster reference associated with a symbolic name. In the implementation, passing a cluster reference to the *register* method is actually passing a proxy-out object reference to the name server (which registers a copy of the proxy -out object). Getting a cluster reference from the name server is actually getting a proxy -out object.

8.6) UML view of the Project

Following UML diagram gives the details about

- Cluster1 class
- Proxy-in-cluster class
- Real cluster1 class
- Interface between them.

All the clusters in this project can have this similar UML structure.

UML Diagram for Cluster1.

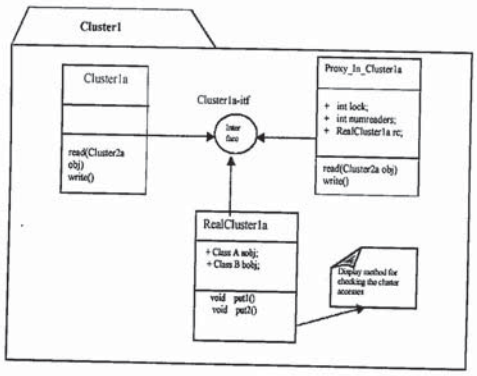

Figure 8.2) Uml Diagram.

This implementation contains three classes. At initial phase all the Classes and interfaces are created in a centralized way, while this concept is adapted to the distributed environment, It will use the RMI registry and binding method. These classes are simple and created for concept establishment.

CHAPTER-9

RESULT REACHED.

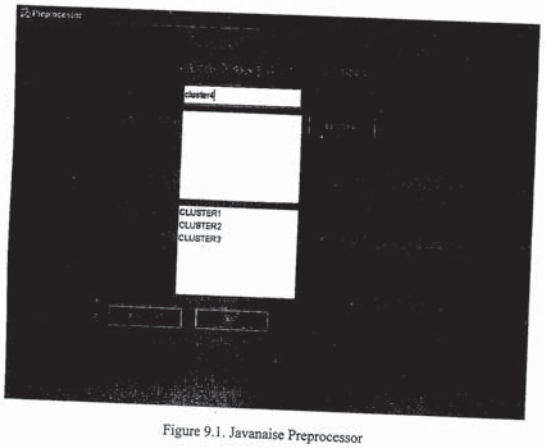

Figure 9.1. Javanaise Preprocessor

The above snapshot shows the creation of cluster in Javanaise server. It is a pre processor, which creates and maintains the clusters .All this operations are carried out by the coordinator.

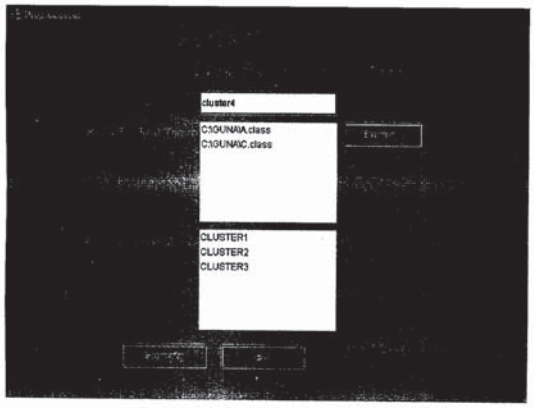

Figure 9.2. Javanaise Preprocessor

Selecting the Real Cluster object by using the "Browse" button. Here we can select the real object from local machine or any machine that can access in network domain.

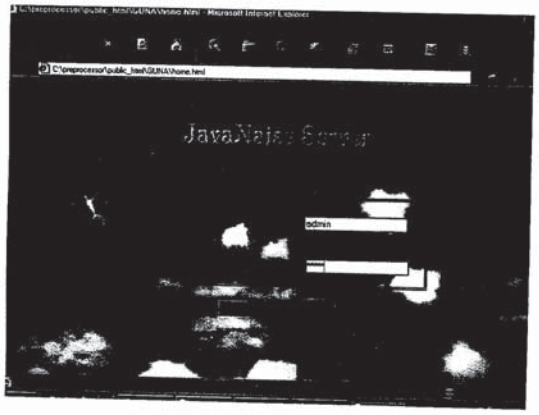

Figure 9.3. Javanaise server

The above snapshot shows the home page of Javanaise Server. In this page all the Javanaise client can login by using their username and password.

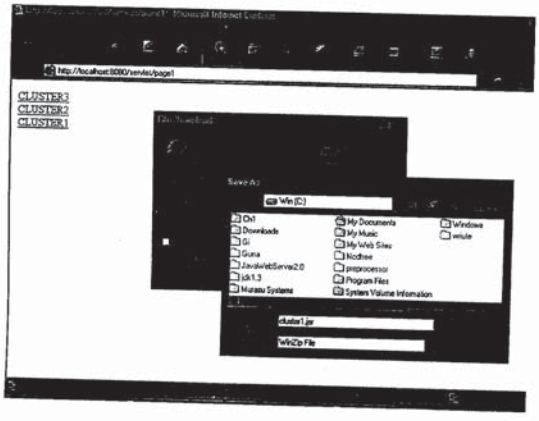

Figure 9.4. Javanaise server component downloading

Once the client entered into Javanaise server the next page shows (above) the entire cluster name and its updation. By clicking the cluster name, client can download the necessary Cluster files into the client side.

Figure 9.5. Javanaise Registry

Start the RMI registry .The above snapshot shows the RMI registry was started.

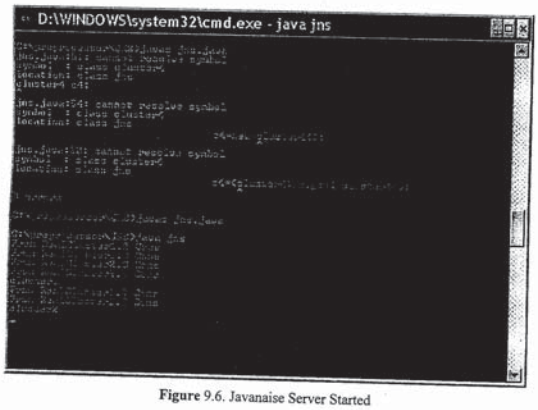

Figure 9.6. Javanaise Server Started

The Above snapshot shows the Javanaise server was started. Once the server is started, it will register all the Real cluster object in the RMI registry. These registered objects are visible to the client side and they can access them by using the registry service.

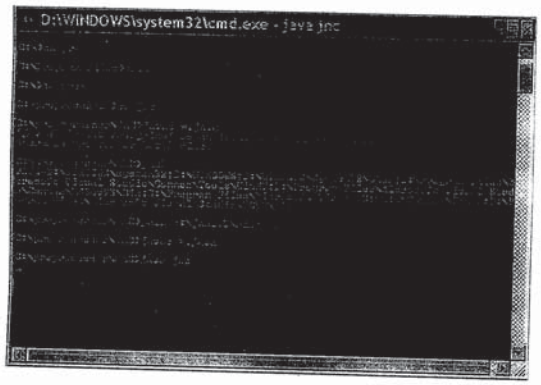

Figure 9.7. Javanaise Client

The above snapshot shows the Javanaise client. By running the jnc program we can start the Javanaise client .

Figure 9.8. Javanaise Client Application Program

The above snapshot shows the Javanaise client application. Here client could know the server's clusters by using the registry service. In the above scenario the client called the one of the method in the server side (first Line). Server executed it and the cluster was transferred to client side. If the client again accesses the same cluster, it could use the cached cluster and display already cached message.

CHAPTER-10

EVALUATION AND RESULTS

To evaluate Javanaise, I implemented the traversal portion of the OO1 benchmark[4], which roughly consists of a traversal of a distributed graph, to compare the benefits provided by the cacheable clusters over Java RMI. Finally, I ported an existing distributed application to Javanaise to demonstrate the adequacy of the platform with a real application.

The experiments were performed on a pair of PC desktops based on Pentium II processors (400 MHz) running Windows NT and the JDK 1.2.2 version of Java VM. These machines were connected through a 100-Mbit Ethernet. All measurements were performed on an isolated network and repeated 10 times; the reported times are the averages of these 10 measurements.

OO1 Benchmark

The traversal portion of the OO1 benchmark (used to evaluate database systems) implements a traversal of a 5,000-node graph in which each node has three children. The traversal is done over seven levels (a total of 3,280 nodes are visited). The nodes are equally distributed on two sites. The children nodes are randomly chosen with a probability of 70 percent that a parent and child are on the same machine. (Note that Javanaise would perform better if this probability were smaller.)

[4] It is a benchmark suit for database.

I have implemented this benchmark with non-cacheable clusters (Java RMI) and cacheable clusters (Javanaise). We did not change the source code, but processed the same code with the Java RMI and Javanaise stub generators. In both cases, the nodes were created on two sites. With Java RMI, a reference to a remote object implies a remote method invocation to that site. With Javanaise cacheable clusters, a reference to a remote object implies that a copy of the object is brought to the local host and the method is invoked locally.

Table 10.1. Performance comparison for the OO1 Benchmark (objects size = 75 bytes).

Invocation mechanism	Time in ms
Java RMI	24
Javanaise (cold)	107.8
Javanaise (hot)	0.57

Table 10.1 presents the results. With a cold start, Javanaise performs three times slower than RMI. This inefficiency has two main causes:

- The object graph has a high percentage of inter object references that are local to one site (70 percent). This implies that when a remote object (on site S2) is brought to the accessing site S1, its children on S2 become remote and will have to be fetched. With Java RMI, when the remote object is invoked (on S2), its children on S2 are local and their invocations are very efficient.
- The variation of the traversal depth of the graph to measure the advantage gained by increasing reuse of objects, shown in the graph. For example, at a depth of 9 levels, objects are reused about 6 times on average. In both cases, the nodes were created on two sites.

The results show that Javanaise can perform better than RMI for the first (cold) travel in the graph depending on the amount of object reuse. In conclusion, this benchmark shows that Javanaise performs within the same order of magnitude as standard Java RMI and can perform better for a general-purpose workload, especially when objects are intensively reused. My plan for future work includes the study of how caching and remote invocation could be combined to take advantage of efficiency related to application structure.

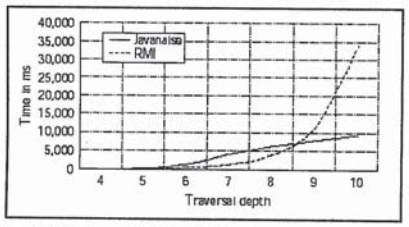

Figure 10.1.Test results for the OO1 benchmark. Javanaise Runs the OO1 benchmark faster than RMI .

CHAPTER-11

CONCLUSION AND PERSPECTIVES

In this paper, I presented my experience with providing a runtime environment for the development of cooperative application on the Internet. In This environment, applications are developed using the Java language, and made available on the Internet through a Web server just like an Applet. Applications are dynamically deployed to client nodes, thanks to Java mobile code, and the Java objects managed by the applications are transparently shared between application instances.

In order to allow for efficient object caching on client nodes, our runtime manages clusters of Java objects, the cluster being the unit of sharing, caching and consistency. Clusters are persistent and stored on disk on the Home Site of the application. Clusters are brought on demand on the client nodes and shared following a specified consistency and synchronization protocol. My claim is that object grouping in clusters can be derived from the application structure by specifying which classes correspond to clusters, other classes defining local objects within clusters.

I have implemented a prototype of the Javanaise runtime, composed of a preprocessor which generates required proxy classes from the interfaces of

cluster classes, and of system classes which manage consistency of cluster replicas cached on client nodes.

The perspectives of this work are first to validate and evaluate this system support through full scale cooperative applications such as the editor mentioned above. Then, this preliminary experiment opens many challenging issues.

REFERENCE

(i). BEA system, (last updated 08/23/2000.) " About Clustering technology in weblogic " , home page, www.e-doc.bea.com,©2001 BEA System inc,

(ii). G.Chockler et al(2000) ,"Implementing Caching Service for Distributed CORBA Objects,"*Proc. IFIP/ACM Int'l Conf. Distributed Systems Platforms and Open Distributed Processing , Lecture Notes in Computer Science*, Springer-Verlag ,Heidelberg,

(iii). Fabian Breg,(2003) "Distributed Scalable Java Operating System for Cluster Architectures" ;Concurrency Practice and Experience.

(iv). D.Hagimont1, D. Louvegnies2, "distributed shared objects for Internet co operative applications", SIRAC Project INRIA, 655 av. Europe, 38330 Montbonnot Saint-Martin, France

(v). D. Hagimont, N. De Palma, F. Boyer, S. Ben Athallat, (2002) 'Improving performances for non-functional properties", Sardes project, F-38334 Saint Ismier, France.

(vi). D. Hagimont et al.(1994) , "Persistent Shared Object Support in theGuide System : Evaluation and Related Work, " *Proc. 9thACM Conf. Object–Oriented Programming, Systems, Languages and Applications* (OOPSLA), pp. 129-144.

(vii). V. Krishnaswamy (2001). ,"Efficient Implementations of Java Remote Method Invocation (RMI)," *Proc. Usenix Conf. on Object Oriented Technology and Systems* (COOTS),Usenix Assn., Berkeley, Calif.

(viii). Pradeep K.Sinha, "Distributed operating Systems"©IEEE Computer society Press,published by Prentice-hall of India privte ltd,India,pp 30-227.

ARULMIGU KALASALINGAM COLLEGE OF ENGINEERING
Accreditated by NBA-AICTE
Anand Nagar, Krishnankoil-626190

Department of Instrumentation & Control Engineering

NCICCA 2005

CERTIFICATE

This is to certify that Mr. / Ms S. GUNASEKARAN has Participated & Presented a paper titled JAVANAISE : for distributed JAVA objects in the National Conference on Intelligent Computing in Communication and Automation organized by the Department of Instrumentation and Control Engineering, Arulmigu Kalasalingam College of Engineering, Krishnankoil during April 1-2, 2005.

INTELLIGENT ENGINEERING

Date : 2-4-2005.
Place: Krishnankoil.

Organizing Chair
NCICCA 2005

(ix). M. Shapiro. "Structure and Encapsulation in Distributed Systems: The Proxy Principle," *Proc. Sixth Int'l Conf. Distributed Computing Systems* (ICDCS), IEEE Computer Society, Los Alamitos, Calif., pp.198– 204.

(x). Sun Microsystems, "Java Remote Method Invocation (RMI)," homepage.http://java.sun.com/products/jdk/rmi/.

(xi). Pradeep K.Sinha. "Distributed operating Systems"©IEEE Computer society Press,published by Prentice-hall of India privte lid,India,pp 30-227.