DESIGN AND IMPLEMENTATION OF A PLUG-IN FRAMEWORK FOR DISTRIBUTED OBJECT TECHNOLOGIES

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

KORAY KADIOĞLU

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

SEPTEMBER 2006

Approval of the Graduate School of Natural and Applied Sciences

Prof. Dr. Canan Özgen
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

Prof. Dr. Ayşe Kiper
Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Dr. Ali Doğru
Supervisor

Examining Committee Members:

Prof. Dr. Adnan Yazıcı	(METU, CENG)	_____
Assoc. Prof. Dr. Ali Doğru	(METU, CENG)	_____
Assoc. Prof. Dr. İsmail Hakkı Toroslu	(METU, CENG)	_____
Assist. Prof. Dr. Pınar Şenkul	(METU, CENG)	_____
Ali Özzeybek	(ASELSAN)	_____

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last name: Koray Kadıoğlu

Signature :

ABSTRACT

DESIGN AND IMPLEMENTATION OF A PLUG-IN FRAMEWORK
FOR DISTRIBUTED OBJECT TECHNOLOGIES

Kadıoğlu, Koray

M.S., Department of Computer Engineering

Supervisor: Assoc. Prof. Dr. Ali Doğru

September 2006, 54 pages

This thesis presents a framework design and implementation that enables run-time selection of different remote call mechanisms. In order to implement an extendable and modular system with run-time upgrading facility, a plug-in framework design is used. Since such a design requires enhanced usage of run-time facilities of the programming language that is used to implement the framework, in this study Java is selected because of its reflection and dynamic class loading facilities. A sample usage of this framework is enabling an application to distribute its tasks over a network using a suitable distributed object technology (DOT). In this work, CORBA, RMI and Java Sockets are the sample DOT plug-ins. A series of performance evaluations of these DOTs are presented to establish a baseline for choosing a suitable DOT for the application domain that uses this framework.

Keywords: Plug-in framework, Java Socket, RMI, CORBA

ÖZ

DAĞITIK NESNE TEKNOLOJİLERİ İÇİN
PLUG-IN ALTYAPISI TASARIM VE GERÇEKLENMESİ

Kadıoğlu, Koray
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi: Assoc. Prof. Dr. Ali Doğru

Eylül 2006, 54 sayfa

Bu tezde, çalışma zamanında değişik uzaktan fonksiyon çağrımı yapabilme mekanizmaları arasında seçim yapılabilmesine olanak tanıyan bir altyapının tasarım ve gerçeklemesi sunulmuştur. Çalışma zamanında yeniden düzenlenebilme yeteneğine sahip, genişleyebilir ve modüler bir sistem gerçeklemek için plug-in altyapısı tasarımı kullanılmıştır. Böyle bir tasarım, gerçeklemede kullanılacak programlama dilinin çalışma zamanına ait yeteneklerine yoğun derecede ihtiyaç duyacağı için, bu çalışmada yansıma ve dinamik sınıf yükleme yetenekleri nedeniyle Java seçilmiştir. Bu altyapının örnek bir kullanımı, herhangi bir uygulamanın çeşitli işleri ağ üzerinde dağıtık nesne teknolojileri ile gerçekleştirebilmesidir. Bu çalışmada CORBA, RMI ve Java soketleri örnek plug-inler olarak gerçeklenmiştir. Bu altyapıyı kullanacak uygulamanın kendine uygun bir dağıtık nesne teknoloji seçebilmesi konusunda bir dayanak oluşturabilmek amacıyla bir dizi performans analizi sunulmuştur.

Anahtar Kelimeler: Plug-in altyapısı, Java Soket, RMI, CORBA

To My Parents and My Lovely Sister

ACKNOWLEDGMENTS

The author wishes to express his deepest gratitude to his supervisor Assoc. Prof. Dr. Ali Doğru for his guidance, advice, criticism, encouragements and insight throughout the research.

The author also thanks to ASELSAN Inc. for the support on academic studies.

The author would also like to thank Emre Başeski for his suggestions and comments.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

DOT	: Distributed Object Technology
OMG	: Object Management Group
CORBA	: Common Object Request Broker Architecture
RMI	: Remote Method Invocation
FSP	: Finite State Process
UDP	: User Datagram Protocol
TCP	: Transfer Control Protocol
IDL	: Interface Definition Language
ORB	: Object Request Broker
JFC	: Java Foundation Classes
JVM	: Java Virtual Machine
XML	: Extensible Markup Language
XSD	: XML Schema Definition
DTD	: Document Type Definition

CHAPTER 1

INTRODUCTION

Responding to different distributed architecture demands for the application domain and choosing a suitable communication methodology for these demands are very important tasks for reducing the complexity of the design architecture and the cost of development life-cycle. By using an extendible design for different communication methodologies, one can reduce the complexity and increase the maintainability of the code. Also, this extendible design requires less testing since only testing the extended parts will be sufficient.

In [1] Michael Pilone explains how to build an extendible design using Java's dynamic class loading mechanism, namely the *plug-in framework*. Plug-in framework constructs the basis for the extendible design mentioned above. With this design, one can add or remove a code segment called *plug-in* to a previously developed application without recompiling or re-testing it. Furthermore, this addition or removal operation does not require the application to be rebooted.

Different communication methodologies called *Distributed Object Technologies* (DOTs) are presented in [2, 3] for the distributed architectures. OMG [2] specifies a DOT called CORBA which enables heterogeneous systems to interact. In [3] two DOTs called RMI and Java Sockets are presented. RMI is specified for only applications that are developed using Java, whereas Java sockets are the generic socket implementations in Java language.

Since different distributed architectures need different DOTs, it is a common practice to choose the suitable one that fits best to the application requirements by using the guidance of previous comparison results of different DOTs. Ahuja and Quintao [4]

evaluate the performance of RMI and Java Sockets. In [5, 6, 7] RMI and CORBA are compared according to their performance based on different applications. A whole performance comparison of all three DOTs is given by Eggen and Eggen in [8].

Thesis Organization

In Chapter 2, plug-in framework development and DOTs are reviewed. The plug-in framework that is implemented in this work is detailed in Chapter 3 which examines different DOTs as different plug-ins to the system. After considering previous comparisons in detail, a new performance comparison is presented in Chapter 4. Finally, in Chapter 5, the presented work is concluded.

Contributions

The contribution of this thesis is implementing a plug-in framework which can use different DOTs as its plug-ins and evaluating the performance of the implemented DOT plug-ins in this framework.

CHAPTER 2

REVIEW OF PLUG-IN FRAMEWORK AND DISTRIBUTED OBJECT TECHNOLOGIES

In this chapter, the plug-in framework design and the distributed object technologies (DOTs) are reviewed. After presenting the pros and cons of developing a plug-in framework, three of DOTs, namely CORBA, RMI & sockets will be explained as the candidate plug-ins of the communication framework implemented in this thesis. Performance results of these DOTs will be discussed and compared with the previous work in chapter 4.

2.1. Plug-in Framework

Every software requires some evolutions in any state of its life-cycle, since the requirements change in time or some bugs may occur in the implementation. Responding to these changes requires either recompilation of the source code and retesting the side effects, or stopping and restarting the system in traditional approaches.

To solve these issues, a new approach called component oriented software development [9] has been developed. Components are code segments that encapsulate functionality and implement a certain interface. Plug-in framework design is simply a component oriented design with extra information that plug-ins are optional rather than required components.

In [10, 11], the advantages of the plug-in framework design are summarized as:

- **Increased extendibility**: Since it is impossible to predict all the future work on the application domain, system should be open to be extended by new

technologies. For instance, a movie player should have the ability to be extended by a new codec. But it is undesirable to build a new version for each codec, so by a plug-in framework design, one can add a new codec to the system without rebuilding a new version.

- **Decomposed large systems:** Large systems require much bigger resources than simple applications, so only required part of the whole should be deployed. This issue can be solved by configuration files but in some cases, handling all these configuration files also becomes a huge problem. In a plug-in framework, only required plug-ins are deployed, so the resources would be utilized. For instance, the Eclipse Integrated Development Environment [12] works with various programming languages but using this plug-in framework design, only the required one is deployed.
- **Run-time upgrading:** In long running safety critical systems, it is a big problem to shut the application down to perform an upgrade. This issue also is solved using the framework, since the plug-ins are run-time configurable.
- **Enhanced third party usage:** Since it is impossible to know all the requirements when it is initially being developed, a system should be open to change as much as possible. These changes should not increase the development and testing cost and effort. In some cases, it is the best way to use specialist third parties. Since it is inconvenient to give all the source code to third parties, building a plug-in framework and giving only the plug-in specification of the system will be sufficient.
- **Increased modularity**: A plug-in simply implements an interface in the framework, so all the information behind is hidden by this encapsulation. This is the primary criteria for system modularization. Any change on such a framework would be easier than through the traditional software design approach.

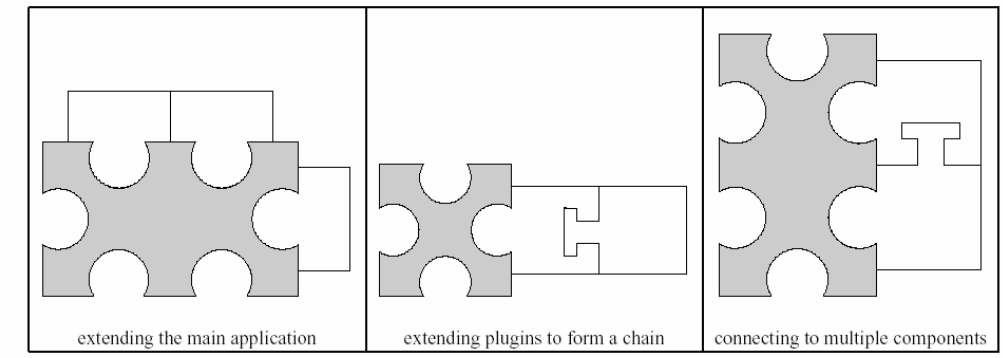
There may also be some problems while developing a plug-in framework, one of which is the desirable architectural changes (component addition, component removal and component replacement) conducted at run-time as stated in [13] by

Oreizy et al. All these issues are connected with the run time facilities that the programming language used to develop the framework supports.

Java programming language has simplified run-time problems associated with plug-in framework development using the run-time interpretation of bytecodes. Using the dynamic class loading mechanism [14] of Java, one can easily start, stop or replace existing plug-ins. This dynamic class loading mechanism and run-time class search semantics can be easily optimized for the application domain by subclassing the ClassLoader given in the Java spec.

In [23], Richard S. Hall discusses a policy-driven class loader namely *ModuleLoader* that subclasses Java's ClassLoader. In addition to the run-time facilities in the Java's ClassLoader that are crucial for a plug-in framework, ModuleLoader also provides support for modifying the search path, managing class versions and adding/removing class definitions at run-time.

Another problem that may arise with plug-in framework development is the dependency issue within the plug-ins. Fitting all the plug-ins in the framework is similar to solving a jigsaw puzzle. Holes represent the plug-in interfaces in the framework, while the pegs are the classes that implement this interface. In Figure 2.1, possible configurations of the plug-ins are stated.



(a) (b) (c)

Figure 2.1: Some possible configurations of plug-ins [10]

In Figure 2.1 (a) and (b), dependency problem would not occur, since (a) has no dependency at all and the dependency at (b) is the responsibility of the plug-in that is added to our plug-in framework. In Figure 2.1 (c), the problem is obvious. A plug-in needs another plug-in to be installed to run correctly.

In [10] this problem is solved using multiple plug-in interfaces and a policy interface called Strategy. By this solution, plug-in framework uses the Strategy class that the plug-in supports to resolve the dependency issue (Figure 2.2). In this design, multiple Strategy classes and plug-in types are supported to avoid multiple dependency problems.

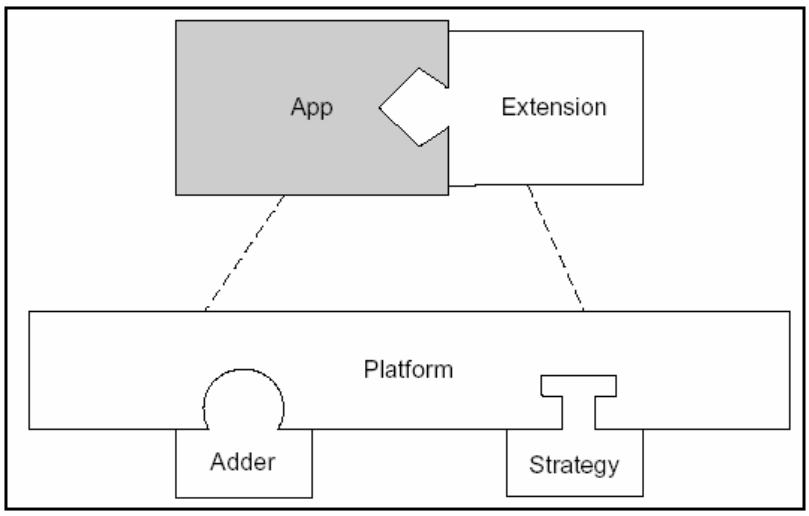


Figure 2.2: Platform architecture managing a two component application [10]

S. Handschuh solves this issue using the Java's reflection in sub plug-in interfaces called OntoPluginServiceProvider and OntoPluginServiceConsumer in [11]. In this design, plug-in framework acts as a rendezvous between provider plug-in that serves a service and the consumer plug-in that needs this service to run correctly.

In [15], this problem is discussed in a behavioral manner. In this approach, it is clearly stated that a group of plug-ins may run correctly, but a new component may cause problems. To solve this, building a structural and behavioral model is needed.

To accomplish this, the Darwin ADL [16] is used in structural modeling whereas the Finite State Process (FSP) [17] is used in behavioral modeling.

2.2. DOTs

Because of the difficulty of handling complex distributed applications, DOTs [18] became the state of the art for distributed systems. In DOTs, all entities are modeled as objects. The goal of such a system is establishing an interaction with remote objects. In this section, three DOTs, namely sockets, *CORBA and RMI* will be reviewed. Detailed information about these technologies can be found in [2, 3, 19].

2.2.1 Sockets

A socket is a combination of an IP address and a port. They first appeared in early UNIX systems in the 1970s and are now the standard low-level communication primitive.

There are two communication protocols that one can use for socket programming: *a)* datagram communication, also known as UDP (user datagram protocol) is a connectionless protocol that requires the local socket descriptor and the receiving socket's address each time a datagram is sent, *b)* the stream communication, also known as TCP (transfer control protocol) is a connection-oriented protocol where one of the sockets listens for a connection request (server), the other asks for a connection (client).

The choice of the protocol strongly depends on the application. Since TCP is connection-oriented, a connection setup time is required. In UDP, there is a size limit of 64 kilobytes on datagrams, while in TCP there is no limit. Also, UDP is an unreliable protocol, that is, there is no guarantee that the order of the datagrams you have sent will be preserved by the receiving socket. On the other hand, TCP is a reliable protocol.

2.2.2 CORBA

A network usually contains heterogeneous elements like mainframes, workstations, PC systems or other kind of hardware and software components. Although this heterogeneous behavior of a network provides satisfying solutions to stand-alone problems, the communication requirement of elements of such a network is a challenging task that cannot be underestimated.

The Object Management Group (OMG) was formed in 1989 to develop, adopt and promote standards for development of applications in distributed heterogeneous environments. The Common Object Request Broker Architecture (CORBA) is one of the first specifications that have been adopted by OMG. The main motivation of CORBA is the object invocation where the objects may reside locally or remotely. A CORBA-based application from any vendor, on any operating system, programming language and network can interoperate with another CORBA-based application.

For each object type, an interface is defined in an IDL file. The IDL interface definition is independent of programming language, but maps to most of the popular programming languages like C, C++, Java, COBOL, Smalltalk, Ada, Lisp, Python, and IDLscript via OMG standards. The IDL file is compiled to generate client stubs and server skeletons for a given language.

The communication between clients and objects is established by a component called *Object Request Broker (ORB)*. When a client wants to invoke an operation on an object that is in the server, the ORB is used by the client to specify the required operation and marshal (serialize) the arguments that will be sent. When the invocation request reaches the server, the same interface is used to unmarshal the arguments. After performing the requested operation, the results are marshaled according to the interface and sent to the requesting client. The last step of the remote operation call is the unmarshalling (read - deserialize) of the result. In Figure 2.3, a sample remote operation call is illustrated.

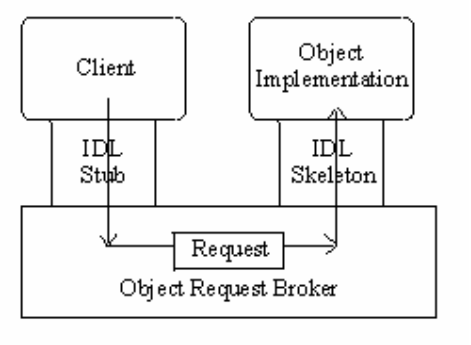


Figure 2.3: A request passing from client to object implementation [2]

2.2.3 RMI

Java Remote Method Invocation (RMI) is a mechanism that allows the invocation of a method on an object that exists in another address space. The address space can be on the same machine or on a different one. Like CORBA, RMI is a remote procedure call mechanism, but unlike CORBA, RMI is not language independent, it is a Java-to-Java technology.

Three processes, *client*, *server* and *object registry*, participate for a remote method invocation. The client is the process that is invoking the method of the remote object; the server is the process that owns the remote object and the object registry is a name server that relates objects with names.

In Figure 2.4, the general architecture of the RMI is presented. The server must first bind its name to the registry. Then, the client looks up the server name in the registry to establish remote references. When a client wants to invoke a remote method, the call is first forwarded to the Stub. The Stub is responsible for sending the remote call to the sever side Skeleton. It opens a socket for remote server, marshals the object parameters and forwards the data stream to the Skeleton. The skeleton contains a method that receives the remote calls, unmarshals the parameters and invokes the actual methods. The results are then marshaled by the Skeleton and sent back to the stub.

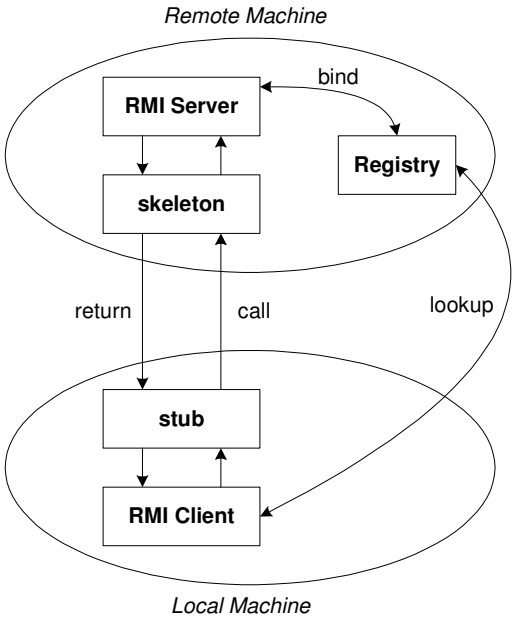


Figure 2.4: A general RMI architecture

CHAPTER 3

DESIGN AND IMPLEMENTATION

In this chapter, the plug-in framework and the sample plug-ins implemented are discussed. After presenting the design and implementation of the framework, plug-ins of three Distributed Object Technologies (DOTs), namely CORBA, RMI & sockets will be explained.

3.1. The Plug-in framework

3.1.1. Plug-in Interface

Plug-in framework shares a common interface called *IPlugin* used for initialization and communication with the plug-ins. This interface is as follows:

```
public interface IPlugin{
        public void start(Object pluginContext, ICommFramework frm);
        public void stop();
}
```

The framework calls *start()* method to load a plug-in and *stop()* method to unload it. It is the responsibility of the plug-in to include all the initialization code in the start method and the cleanup code in the stop method.

3.1.2. Sharing Data with a Plug-in

In this plug-in based approach, the most important decision is determining the portion of the application, which will use this plug-in framework, that is shared with a plug-in. The start method takes this portion as an argument which is passed by the

application to the plug-in framework. The definition of this class is in the application, so to make the plug-in framework independent from this class; it is passed as an *Object* to the plug-in. In Figure 3.1 the dependency graph of the design is shown.

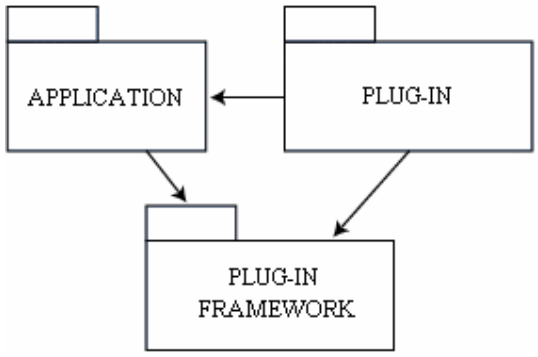


Figure 3.1: The dependencies of the plug-in framework design

3.1.3. Plug-in Framework Interface

Another parameter to the start method of the plug-in is the interface of the framework (*ICommFramework*). Using this interface, a plug-in may use some features supported by the framework. In Figure 3.2, a use-case diagram based on this interface is shown. A sample interface is as follows:

```
public interface ICommFramework {
        public void startPlugins();
        public void stopPlugins();
        public void startPlugin(File pluginPath);
        public void stopPlugin(String pluginClassName);
        public Object[] getPlugins();
        public Object getPlugin(String pluginClassName);
}
```

A plug-in may obtain another plug-in class loaded previously from the framework over this interface. Also, the application using this framework can use these methods to operate any task on all the plug-ins to the system.

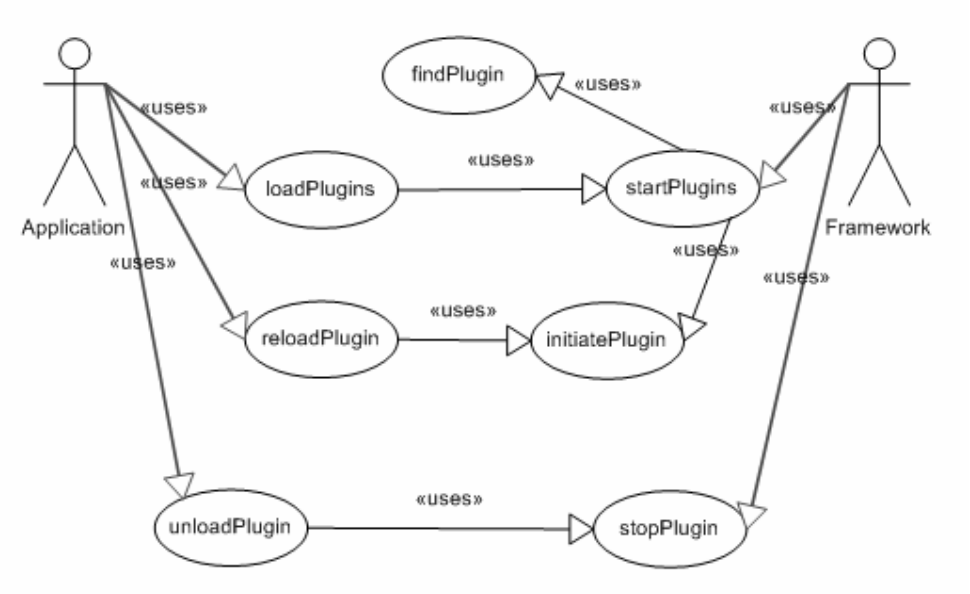


Figure 3.2 Use-case diagram of the framework design

3.1.4. Packing a Plug-in

For any plug-in, there may be more then one class to support the extra functionality to the application. To bundle these classes, a JAR file is used. A JAR file is the file that includes many java classes inside. Java spec contains many classes to support processing JAR files. By using these classes, the name of the plug-in class that implements *IPlugin* interface can be obtained by the framework to use in the reflection mechanism from the manifest file of the JAR package.

The manifest file is a special file that contains information about the class files packaged in a JAR file. By tailoring this "meta" information that the manifest contains, you enable the JAR file to serve a variety of purposes. There can be only one manifest file and it always has the path *META-INF/MANIFEST.MF*. The manifest file contains header – value pairs separated by a colon.

In this work it is assumed that the manifest file includes an attribute called *Plugin-Class* that includes the name of the main plug-in class to be loaded by the framework. A sample manifest is as follows:

```
Manifest-Version: 1.0
Plugin-Class: com.sample.SamplePlugin
```

In this manifest, it is stated that the class file in this JAR package that implements the *IPlugin* interface is called *SamplePlugin* and it is located in *sample* package which is also located in the *com* package within the JAR file.

3.1.5. Locating a Plug-in

The plug-in framework locates the plug-ins by a *java.io.File* parameter that indicates the folder containing plug-ins, passed by the application that uses this framework. By extending *java.io.FileFilter* to filter only the JAR files and passing it as a parameter to *File.listFiles(FileFilter)* method, framework lists all the JAR files in the specified directory. Using one of the standard Java Foundation Classes (JFC) *java.util.JarFile*, the framework reaches the manifest (*JarFile.getManifest()*) and the attributes inside the manifest (*Manifest.getMainAttributes()*).

3.1.6. Run-time Class Loading

To load any class at run time, JFC provides *java.lang.ClassLoader*. The plug-in framework loads the plug-in class in a JAR file by the *java.net.URLClassLoader* by passing the name of the JAR file obtained previously as a parameter sent to it. Using the JVM class loader as the parent to the new classloader ensures that the plug-in class can find any class in the application classpath in addition with the plug-in related classes. The run-time class loading mechanism is as follows:

```
...
//jarFile: the java.io.File that represents the plug-in JAR file
URL[] urls = new URL[] { jarFile.toURL()};

URLClassLoader loader = new URLClassLoader(urls);

//className: the class name implementing Iplugin found in manifest
Class pluginClass = loader.loadClass(className);

Object pluginInstance = pluginClass.newInstance();
IPlugin foundPlugin = (IPlugin) pluginInstance;
...
```

3.1.7. Dependency Problem within the Plug-ins

While loading the plug-ins found in the directory specified by the application that uses this plug-in framework, it is possible that a dependency problem may occur within the plug-ins. One plug-in may require another plug-in to be loaded previously to run correctly. To solve this problem, the manifest file attributes have been populated. Another attribute called *Plugin-Dependencies* states the dependencies of the plug-in to the framework as follows in a sample manifest:

```
Manifest-Version: 1.0
Plugin-Class: com.sample.SamplePlugin
Plugin-Dependencies: sample1.SimplePlugin,sample2.ComplexPlugin
```

This sample manifest states that this plug-in requires some classes within the scope of two other plug-in JAR files which include *SimplePlugin* and *ComplexPlugin* classes that implement the *IPlugin* interface. By using *Plugin-Dependencies* attribute, the framework obtains the dependency graph of the plug-ins in the specified directory before loading them. After this graph is constructed, framework starts loading the plug-ins to the system accordingly. Cyclic dependencies are handled, so a dependency problem is avoided.

The new classloader that is used to load the plug-ins take the JVM classloader of the application as the parent parameter to make application classes visible to the plug-in. Rather than using new classloader instance for each plug-in, only one classloader is instantiated in the framework using all the JAR files of all the plug-ins as the URL parameter, so that a plug-in can obtain any class in any of the plug-ins.

3.1.8. Defining Additional Constraints on Plug-ins

The plug-in framework also supports additional interfaces that are defined in the application that uses this framework which force all the plug-ins to satisfy additional requirements. This task is performed by the reflection mechanism of the Java spec as follows:

```
...
Class iFace = Class.forName(interfaceName);
boolean b = iFace.isAssignableFrom(pluginInstance.getClass());
İf (b){
        //the check is ok!
}
```

Application passes the names of the interfaces to the plug-in framework, and in this code segment, the framework checks whether the loaded plug-in has implemented that interface or not.

3.1.9. Using the Framework for DOTs

This plug-in framework design is as generic as possible. It can be used to perform any kind of plug-in based extension of any application. A sample use of the framework for DOTs is shown in Figure 3.3 with a sequence diagram. The code segment to use this framework by an application is as follows:

```
...
CPluginContext context = new CPluginContext();

File[] pluginPath = new File[1];
pluginPath[0] = new File("libraries/plugins");

String[] constraintInterface = new String[1];
constraintInterface[0] = "com.sampleApp.ICommPlugin";

ICommFramework commFramework =
        new CCommFramework(
                context,
                pluginPath,
                constraintInterface);
commFramework.startPlugins();
...
```

CCommFramework is the main framework class that implements *ICommFramework* interface that defines the framework to the application that uses it and to the plug-ins that it has been started. Explanations for the parameters of the constructor are:

- **context:** The portion of the application desired to be shared with the plug-in
- **pluginPath:** The paths in which the framework will try to locate the plug-ins
- **constraintInterface:** The names of the interface classes that the application wants to force its plug-ins to implement.

The *CPluginContext* class and *ICommPlugin* interface is as follows:

```
public class CPluginContext{
        public void receiveStringMessage(String message){
                ...
        }
        public void receiveIntegerMessage(int message){
                ...
        }
        ...
}
```

```
public interface ICommPlugin{
        public void sendStringMessage(String message);
        public void sendIntegerMessage(int message);
        ...
}
```

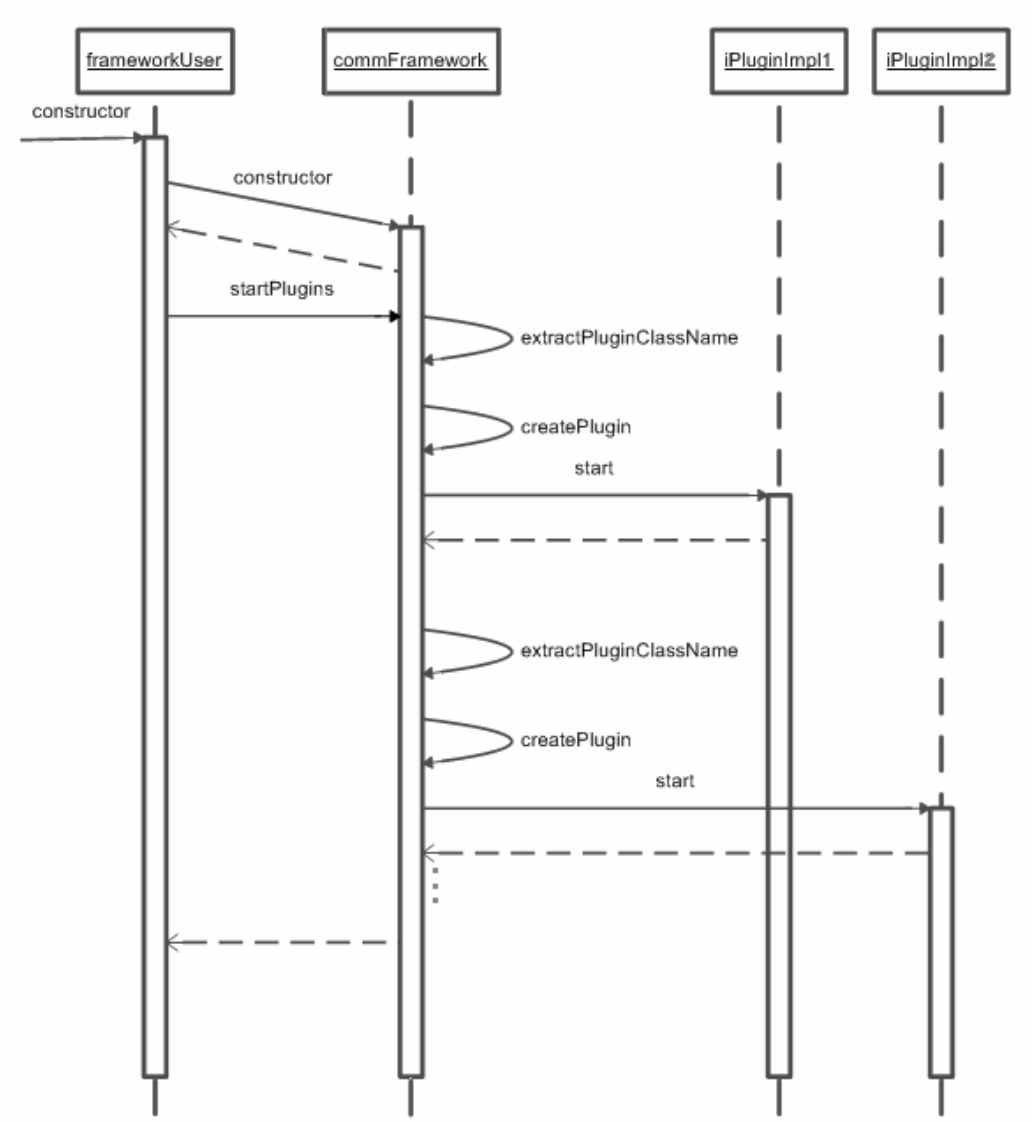


Figure 3.3 Sequence diagram of using this framework

In *CPluginContext* class, application defines the outer interface behavior that is responsible for performing the related task when a message receives. In *ICommPlugin* interface, the application forces the plug-ins to have outer interface behavior also. By implementing this interface, plug-in becomes the responsible part of sending the related message using the related DOT.

The application will use the plug-in framework for sending a string message over related DOT as follows:

```
...
//commFramework: the plug-in framework of type ICommFramewok
Object[] plugins = commFramework.getPlugins();
for (int i=0; i<plugins.length; i++){
        ICommPlugin plg = (ICommPlugin) plugins[i];
        plg.sendStringMessage("Hello World");
}
...
```

Using *ICommFramework* interface, application gets all the loaded DOT plug-ins and sends the related message over all of them. If only one plug-in is desired, the application can obtain related plug-in using *getPlugin()* method with the name of the related plug-in class name as the parameter.

When the application needs to change the DOT at run-time or want to upgrade the related DOT plug-in, it stops the undesired one first using the main class of the plug-in passed as a parameter to method *stopPlugin()*. Then application can start the new plug-in using the method startPlugin() using the main class of the new plug-in passed as a parameter to it. This usage of the framework is shown in Figure 3.4 by a sequence diagram.

Using the definitions above, any of the plug-ins will look like in this sample application as follows:

```
public class SamplePlugin implements IPlugin, ICommPlugin{
        CPluginContext context;
        public void sendStringMessage(String message){
                //the code responsible of sending the message
        }
        public void sendIntegerMessage(int message){
                //the code responsible of sending the message
        }
        public void start(Object pluginContext, ICommFramework frm){
                this.context = (CPluginContext) pluginContext;
                //the initialization code
        }
        public void stop(){
                //the cleanup code
        }
        //the method that is called when a message is received
        public void messageReceived(Object message){
                //parsing code to analyse the message
                ...
                //pass the message inside the application
                context.receiveStringMessage(str);
        }
}
```

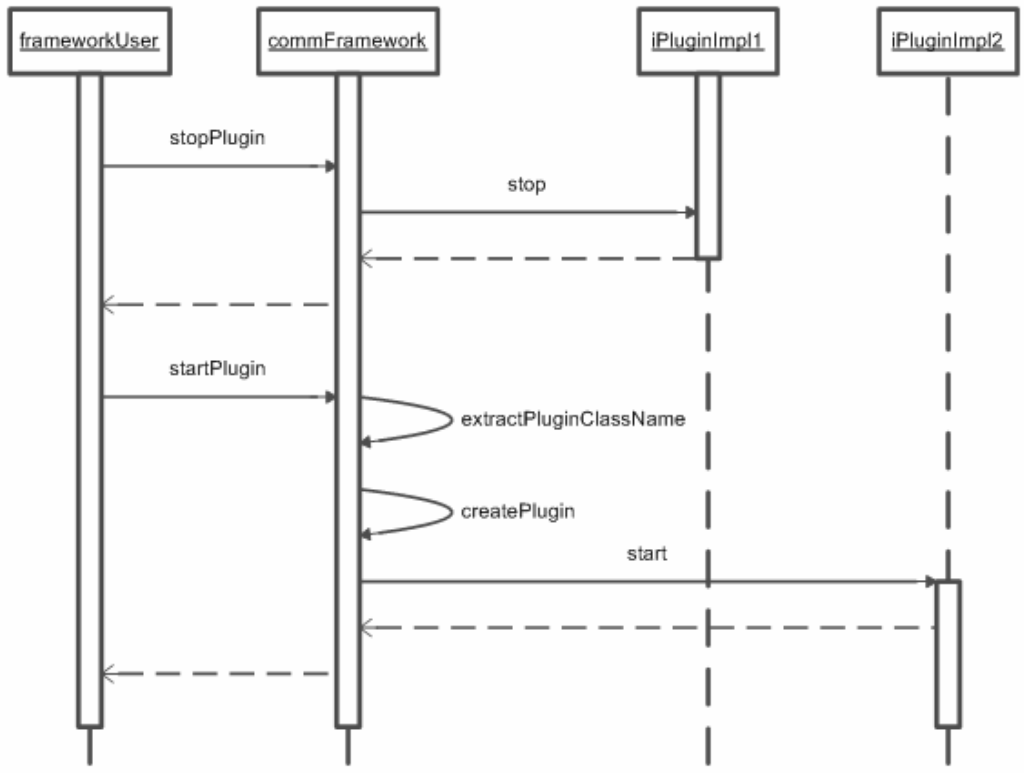


Figure 3.4 Sequence diagram of run-time plug-in loading/unloading

Since the SamplePlugin class both implements *IPlugin* and *ICommPlugin* interfaces, the plug-in framework will accept it as a plug-in to the system. When a message is received to this plug-in, it parses the message to forward to the application using the right method of the *CPluginContext* class defined by the application.

3.2. Socket Plug-in

In this section, since the socket plug-in uses xml data as a message over a Java socket, before discussing the plug-in itself, extensible markup language (XML), XML Schema definition (XSD) and the Castor tool (for the auto-generation of the XSD based Java classes) are explained briefly.

3.2.1. Extensible Markup Language (XML)

XML is used to define documents with a standard format that can be read by any XML-compatible application. XML itself is not a markup language. Instead, it is a "metalanguage" that can be used to create specific markup languages.

While XML is commonly used in Web applications, many other programs can use XML documents as well. For instance, computer systems and databases contain data in incompatible formats. Converting the data to XML can greatly reduce this complexity and create data that can be read by different types of applications. A sample XML file is as follows:

```
<?xml version="1.0"?>
<rootNode>
        <childNode>some information</childNode>
</rootNode>
```

First line of the document defines the XML version of the document and should always be included. All XML documents must have a single tag pair to define the root tag, in which all the sub-tags are nested. XML tags are case-sensitive and since all the tags should have a closing tag also, they should be properly nested.

3.2.2. XML Schema Description

An XML schema is a description of a type of XML document, typically expressed in terms of constraints on the structure and content of documents of that type. An XML schema provides a view of the document type at a relatively high level of abstraction.

There are languages developed specifically to express XML schemas. The *Document Type Definition (DTD) language* is a schema language that is of relatively limited capability, on the other hand two other more expressive XML schema languages are *XML Schema (XSD)* and *RELAX NG.* In this thesis, XSD files are used as the schema for the XML files. An XSD file is written in the W3C XML Schema language [20].

The process of checking to see if an XML document conforms to a schema is called validation. All XML documents must be well-formed, but it is not required that a document be valid unless the document is also checked with its associated schema.

An example XML Schema file (*person.xsd*) and the corresponding XML file (*person.xml*) associated with that schema are as follows:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
        <xs:element name="person" type="Person"/>
        <xs:complexType name="Person">
                <xs:sequence>
                        <xs:element name="name" type="xs:string"/>
                        <xs:element name="age" type="xs:int"/>
                </xs:sequence>
        </xs:complexType>
</xs:schema>
```

```
<person
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="person.xsd">
        <name>Koray</name>
        <age>25</age>
</person>
```

3.2.3. Generating Java Classes from XSD File

In order to use XML data inside Java code of the socket plug-in, Castor libraries are used. Castor provides the only open-source Schema Object Model that loads an XML Schema in a Java representation. It also generates Java classes given an XML Schema and performs validation. More information on this tool can be found in [21]

3.2.4. The Plug-in Design and Implementation

Using the *CPluginContext* class and the *ICommPlugin* interface defined in 3.1.9, related sample schema definition file can be defined as follows:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="MessageType">
    <xs:complexType>
      <xs:choice>
        <xs:element name="OuterSpaceToApp">
          <xs:complexType>
            <xs:choice>
              <xs:element
                    name="StringMessage" type="xs:string"/>
              <xs:element
                    name="IntegerMessage" type="xs:int"/>
            </xs:choice>
          </xs:complexType>
        </xs:element>
        <xs:element name="AppToOuterSpace">
          <xs:complexType>
            <xs:choice>
              <xs:element
                    name="StringMessage" type="xs:string"/>
              <xs:element
                    name="IntegerMessage" type="xs:int"/>
            </xs:choice>
          </xs:complexType>
        </xs:element>
      </xs:choice>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

In Figure 3.5, a sample graphical representation of the schema file above is given. Since this schema file is to be used for defining the message type incoming or outgoing; the important part of it is defining sub elements as choice type till the leaf elements. Using choice types, those are the "OR" bubbles in the figure, ensures that only one type of a message is used at a time.

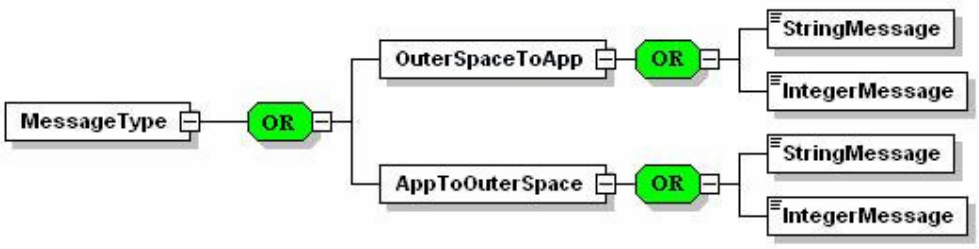


Figure 3.5: Graphical representation of the schema definition file

The main class of the socket plug-in, namely the *CSocketPlugin*, implements both *IPlugin* and *ICommPlugin* interfaces as the *SamplePlugin* class given in 3.1.9. *CSocketPlugin* methods are responsible of parsing the XML data incoming to pass it to the plug-in framework using appropriate *CPluginContext* methods and generating appropriate XML data defined by the XSD file and passing it to the Java socket.

To validate and examine xml data inside *CSocketPlugin*, auto generated classes of the schema definition file is used. This auto-generation is done by Castor tool. Xml schema support of Castor is defined in [21]. After adding these xsd definition classes to the plug-in JAR package, using xml data as a message over Java sockets is done using the Castor generated special methods *marshall()* and *unmarshal()*. *marshall()* method is responsible of serializing schema based auto-generated classes to the given output. In this case, output is the output stream of the socket connection. *unmarshal()* method is the method used for deserialization (read) from the given input. In this case, input is the input stream of the socket connection. Since the Castor generated *unmarshal()* method could not understand the end of an xml message by itself, rather than using the input stream directly, parsing the xml data is handled by the socket plug-in classes before passing it to the *unmarshal()* method.

3.3. CORBA Plug-in

In this section, since the CORBA plug-in uses interface definition language (IDL) to define the clients over interfaces, before discussing the plug-in itself, IDL syntax and *JacORB* object request broker (ORB) that is used in this work are explained briefly.

3.3.1. Interface Definition Language (IDL)

The OMG IDL supports the specification of object interfaces. In IDL, the objects itself are not defined, so the implementation of this interfaces can be done in a standard programming language. Currently, the OMG has standardized on language bindings for the C, C++, Java, Ada, COBOL, Smalltalk, Objective C, and Lisp programming languages. Since the clients only depend on interfaces, heterogeneous systems can communicate.

By using OMG IDL, one can describe object interfaces, namely the operations, attributes of basic and complex data types, and exceptions that may rise. A sample IDL file is as follows:

```
module SampleModule{
  struct SampleStruct{
    string name;
    long id;
  };

  Exception SampleNotFoundException{};

  interface ISample{
    readonly attribute string description;
    SampleStruct getSample() raises(SampleNotFoundException);
  };

  interface ISampleFactory{
       ISample createSample(in string description);
  };
};
```

In this IDL file, a sample module named *SampleModule* is defined. A module defines a scope for the inner definitions. *SampleModule* contains a data structure (*SampleStruct*), an exception (*SampleNotFoundException*) and two interfaces (*ISample* and *ISampleFactory*).

Also note that the parameters to operations are tagged with the keywords *in*, *out*, or *inout*. The *in* keyword indicates the data are passed from the client to the object. The *out* keyword indicates that the data are returned from the object to the client, and *inout* indicates that the data is passed from the client to the object and then returned back to the client.

IDL declarations are compiled with an IDL compiler and converted to their associated representations in the target programming languages according to the standard language binding. For instance, an IDL *module* is mapped to a Java *package* when compiled.

3.3.2. Object Request Broker (ORB)

The Object Request Broker (ORB) is responsible from communication between remote objects. It locates the remote object, passes the request, waits for results and then passes the results back to the client.

Using ORB to communicate with a remote object results in location transparency, namely the client could not understand exact place the remote object is located. Also ORB implements programming language independence for the request, so heterogeneous systems can communicate using the language bindings stated in 3.3.1.

There are many ORB vendors. Since CORBA 2.0 defines a network protocol called *IIOP* (Internet Inter-ORB Protocol), no inter-communication problem does occur. In this work, an open source ORB called *JacORB* is chosen. More information on JacORB can be found in [22].

3.3.3. The Plug-in Design and Implementation

Using the *CPluginContext* class and the *ICommPlugin* interface defined in 3.1.9, related IDL files (plug-in-side and the connected application-side) can be defined as follows:

```
module MCORBAPlugin{
  interface CORBAPlugin{
    void receiveStringMessage(in string stringMessage);
    void receiveIntegerMessage(in long intMessage);
  };
};
```

```
module MOuterSpace{
  interface OuterSpace{
    void receiveStringMessage(in string stringMessage);
    void receiveIntegerMessage(in long intMessage);
  };
};
```

MCORBAPlugin module is defined for the CORBA plug-in implemented in this work. *MOuterSpace* module describes the application that the application using the plug-in framework implemented in this thesis desires to communicate with. Since the sample definitions in 3.1.9 are symmetric, these IDL definitions are very similar. After compiling these IDL files, all the classes generated are added into the CORBA plug-in JAR package.

Since Java spec has CORBA support, to use another ORB rather than the Java ORB, some Java Virtual Machine (JVM) arguments are required. To specify the path of the JacORB libraries to the JVM, following arguments can be used:

```
-Djava.endorsed.dirs="d:\CORBA\JacORB\lib"
-Dorg.omg.CORBA.ORBClass=org.jacorb.orb.ORB
-Dorg.omg.CORBA.ORBSingletonClass=org.jacorb.orb.ORBSingleton
-Dcustom.props=.\jacorbFramework.properties
```

In these arguments, the library paths of the JacORB are introduced to the VM with another path of a file named "*jacorbFramework.properties*" as the custom properties file. This file is used by the ORB to define custom properties such as the port to open, default logging configurations or the proxy addresses.

The main class of the CORBA plug-in, namely the *CCORBAPlugin*, implements both *IPlugin* and *ICommPlugin* interfaces as the *SamplePlugin* class given in 3.1.9. *CCORBAPlugin* also extends the *CORBAPluginPOA* abstract class that includes *CORBAPlugin* interface operations defined in the IDL file which are auto-generated by the IDL compiler. All these methods only pass the incoming message to the appropriate *CPluginContext* methods.

For the outgoing messages, the methods included in the *ICommPlugin* interface forward the messages to the appropriate methods defined in the *OuterSpace* interface in the IDL file. Since this IDL file has been compiled, *CCORBAPlugin* class gets these methods over the name service using the ORB and the *OuterSpaceHelper* class that is auto generated by the IDL compiler as follows:

```
...
//gets the object from the name service
//serverName: the name of the server-side remote object
Object obj = getObjectFromNS(serverName);

OuterSpace os = OuterSpaceHelper.narrow(obj);
...
```

The method *getObjectFromNS()* method includes the ORB initialization codes to obtain the object using its name from the naming service that is already being running by the command line command "*ns*". After this object is returned successfully from this method, *CCORBAPlugin* class narrows it to the IDL generated interface *OuterSpace*. After this, any method of this interface can be called over the name service for forwarding the messages which comes from the application that uses this framework.

3.4. RMI Plug-in

Using the *CPluginContext* class and the *ICommPlugin* interface defined in 3.1.9, related server interfaces of both sides can be defined as follows:

```
public interface RMIPluginServer extends Remote{
        public void receiveStringMessage(String message)
                throws RemoteException;
        public void receiveIntegerMessage(int Message)
                throws RemoteException;
}
```

```
public interface OuterSpaceServer extends Remote{
        public void receiveStringMessage(String message)
                throws RemoteException;
        public void receiveIntegerMessage(int Message)
                throws RemoteException;
}
```

The important thing in these interfaces is they both extend *java.rmi.Remote*. This gives the ability of registering these classes to the RMI naming service for remote usage by any client connected to that naming service. RMI naming service can be started by the command line command "*start rmiregistry*".

Since method call over naming service means serialization of objects over the network, all the parameters of the operations defined in these interfaces must extend *java.io.Serializable*. Since *String* definition already extends it and *int* is one of the primitive types, in these interface definitions there is no need to define a new class extending *java.io.Serializable.*

For the security issues, making a connection over RMI requires definition of a policy file. In this file one can define the socket permissions that the naming service needs. A sample policy file named "*java.policy*" can be defined as follows:

```
grant {
   permission java.net.SocketPermission "*:1024-65535",
      "connect,accept";
   permission java.net.SocketPermission "*:80", "connect";
};
```

The main class of the RMI plug-in, namely the *CRMIPlugin*, implements both *IPlugin* and *ICommPlugin* interfaces as the *SamplePlugin* class given in 3.1.9. *CRMIPlugin* also extends *java.rmi.UnicastRemoteObject* and also implements the *RMIPluginServer* interface which is defined above to enable remote method invocation. All these methods defined in the *RMIPluginServer* interface only pass the incoming message to the appropriate *CPluginContext* methods.

For the outgoing messages, the methods included in the *ICommPlugin* interface forward the messages to the appropriate methods defined in the *OuterSpaceServer* interface. *CRMIPlugin* class gets these methods over the name service as follows:

```
// set the security manager for the client
System.setSecurityManager(new RMISecurityManager());

//get the remote object from the registry
try{
        //url: url of the naming Service with the server name
        String url = "//localhost/SAMPLE-SERVER";
        //narrow the object down to a specific one
        OuterSpaceServer remoteObject =
              (OuterSpaceServer)Naming.lookup(url);
        //use the remote object to call any method
        ...
} catch (RemoteException exc){
        System.out.println("Error in lookup: " + exc.toString());
} catch (java.net.MalformedURLException exc){
        System.out.println("Malformed URL: " + exc.toString());
} catch (java.rmi.NotBoundException exc){
        System.out.println("NotBound: " + exc.toString());
}
```

The important thing in this code segment is that the plug-in narrows the object returned from the naming service to the server class of the connected application using the interface *OuterSpaceServer* defined in that application. Also the connected application should know the *RMIPluginServer* interface to call a method over naming service. This causes a circular dependency problem, and the solution is separating interface definitions from the local class implementations. Figure 3.6 shows the resolution of this dependency problem.

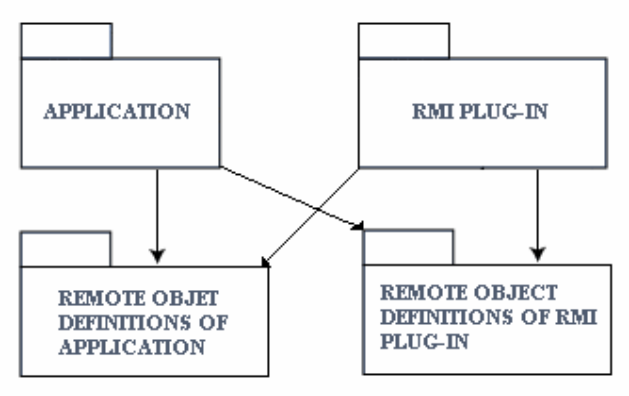


Figure 3.6: Resolution of the circular dependency problem in RMI

To make the class that extends *java.rmi.Remote* visible to the clients over the naming service, after implementing the *RMIPluginServer* interface in *CRMIPlugin* main class of the RMI plug-in, compiling it with a specific compiler named "*rmic*" is required. The result of this compilation is a new class named "*CRMIPlugin_Stub*".

To use RMI naming service, some Java Virtual Machine (JVM) arguments are required. To specify the path of the stub classes and the policy file to the JVM, following arguments can be used:

```
-Djava.rmi.server.codebase=file:/D:\projects\eclipse\rmiPlugin/
-Djava.security.policy=java.policy
```

CHAPTER 4

EVALUATION OF COMPARISON RESULTS

In this chapter, the series of performance evaluations on three Distributed Object Technologies (DOTs) namely CORBA, RMI and Java Sockets are explained. After presenting the testing environment, the response times of different method call types are discussed.

4.1. Testing Environment

In the experiments that are detailed in the following sections, all Intel based PC's consisting of single 2.60 GHz processor, 1.50 GB RAM, running Windows XP Service Pack 2 that are connected by 100 Mbps Fast Ethernet has been used.

At the tables in the following sections, data represents the response time of each method type that is called 1000 times. For the experiments, two different strategies have been followed: (a) the server and the client codes have been executed on a single computer and (b) the execution has been distributed on different computers one of which is the server and the other is the client. Each experiment is mean of 100 runs in which the response time data has been obtained using *getCurrentTimeMillis()* method of Java spec as follows:

```
…
long startTime = System.currentTimeMillis();
//test code for 1000 times of that method call
…
long finishTime = System.currentTimeMillis();
long executionTime = finishTime – startTime;
…
```

4.2. General Discussions

At all the experiments performed in the following sections, following aspects regarding to the response times were identified:

(i) The response times of all DOT plug-ins when the client and the server codes executed on a single computer are greater than the response time results when client and server executions are distributed to different computers. The reason for the slowness in single computer run is that the two instances of Java Virtual Machine (JVM), that are associated with client and server respectively, run concurrently which causes competition for resources and CPU scheduling, which is detailed in [6]. This causes extra delay rather than the multiple computers run in which there is less competition since each JVM has its own CPU and all the delay is generated by the network transmission.

(ii) The drop in the response times of the Socket plug-in when multiple computers are used compared with the single computer run is much greater than other DOT plug-ins. This shows that Socket plug-in requires much resource than the others, since the drop is caused by the decrease in competition for resources when two JVMs work in different computers

(iii) Method calls with void return type have less response times compared with the method calls with return types same as parameter types. This is because of an extra marshal – unmarshal delay caused by the returned value.

4.3. Parameterless Call of Return Type Void

In this experiment, a method call with no parameters and void return type has been evaluated. In Table 4.1, the response time results have been stated. In addition to the general discussions stated in section 4.2, it is identified that the response time of RMI plug-in is less than CORBA plug-in, which is also less than the Socket plug-in in both single computer and multiple computers run. But compared with the difference in the response times of CORBA and RMI plug-ins, Socket plug-in can be counted as fast as the CORBA plug-in in the multiple computers run since the difference is much less compared with the difference between CORBA and RMI.

Table 4.1: Results for no parameter

	CORBA	RMI	SOCKET
Single Computer	515,4	313,9	1922,9
Multiple Computers	401,4	274,9	425,7

4.4. Call of Return Type Void with Primitive Type Parameter

In this experiment, a method call with a primitive type parameter and void return type has been evaluated. For simplicity, *int* has been chosen for the primitive type with the value 0. In Table 4.2, the response time results have been stated. In addition to the general discussions stated in section 4.2, the following aspects regarding to the response times were identified:

(i) The response time of RMI plug-in is less than CORBA plug-in, which is also less than the Socket plug-in in both single computer and multiple computers run. But compared with the difference in the response times of CORBA and RMI plug-ins, Socket plug-in can be counted as fast as the CORBA plug-in in the multiple computers run since the difference is much less compared with the difference between CORBA and RMI.

(ii) The response times of all DOT plug-ins are similar for a method call with no parameter, which are stated in Table 4.1, and a method call with a single *int* parameter. This is because the serialization of primitive types requires negligible resource.

Table 4.2: Results for *int* parameter

	CORBA	RMI	SOCKET
Single Computer	501,4	320,2	2052,6
Multiple Computers	396,7	273,4	423,8

In Figure 4.1, average response time results of all primitive data types are stated. As compared with the results obtained in this study, it is identified from the Figure 4.1 that CORBA has greater latency than RMI in both single computer and multiple computers run, which is consistent with the results in Table 4.1 that are obtained in this work. In Table 4.1, response times for primitive data types decreased at multiple computers run due to two different JVMs running on two different computers. On the

other hand, in Figure 4.1, a slight increase is stated at the same condition. This is because in [7], the programming language chosen for the testing environment is C++ which does not require a virtual machine to run on a computer.

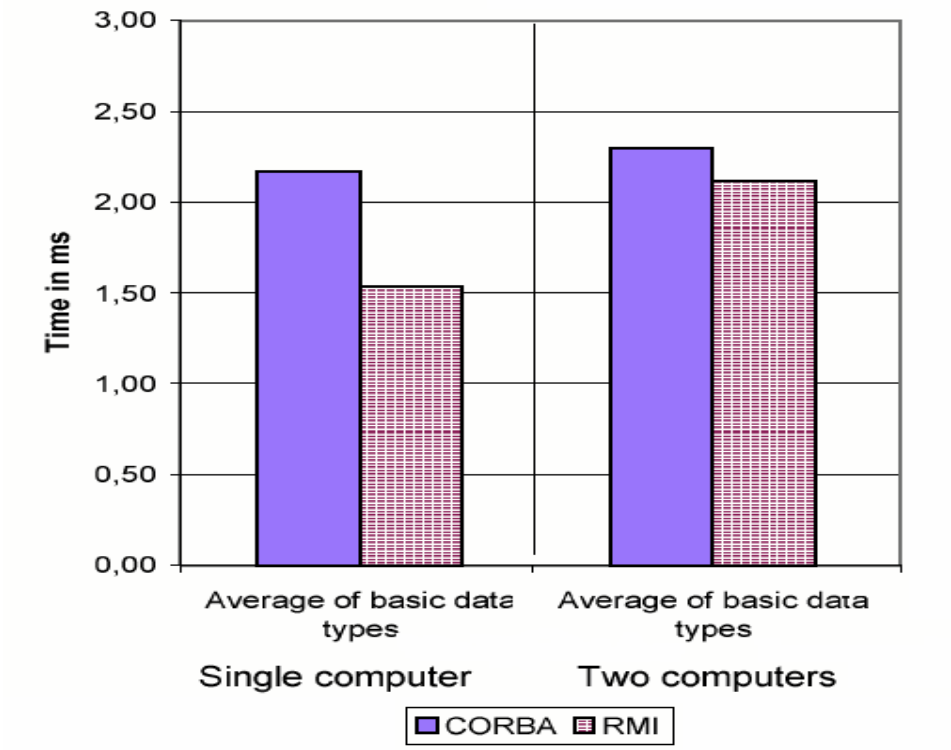


Figure 4.1 Average results of primitive data types from [7]

4.5. Call of Return Type Void with Varying Length Parameter

In this experiment, a method call with a varying length parameter and void return type has been evaluated. For simplicity, *String* has been chosen for the varying length parameter type with value of "a" characters of varying length. In Table 4.3, Table 4.4, Figure 4.2.and Figure 4.3, the response time results have been stated for single computer and multiple computers run. The test cases are populated using the length of the *String* data. In addition to the general discussions stated in section 4.2, the following aspects regarding to the response times were identified:

(i) The response time of RMI plug-in is less than CORBA plug-in in both single computer and multiple computers run and in all the length variations.

(ii) The response time of the Socket plug-in is much greater than other DOT plug-ins in single computer run and in all the length variations.

(iii) The response time of Socket plug-in in multiple computers run is the fastest one compared with the other DOTs till the *String* length reaches up to 10000 characters. This is because the increase in the *String* length is similar with the increase in the *int* primitive type in the Socket plug-in. Since there is no need for extra tag elements in the XML data to express length variations, using XML data works fine.

(iv) The response times of RMI and CORBA plug-ins for a method call with a single *String* parameter are greater than the response times for a method call with a single *int* parameter, which are stated in Table 4.2, in both single computer and multiple computers run. This is because the serialization of primitive types requires less resource compared with the serialization of the Java objects.

(v) In single computer run, the response times of Socket plug-in for a method call with a single *String* parameter is much greater than the response times for a method call with a single *int* parameter, which are stated in Table 4.2.

(vi) In multiple computers run, the response times of Socket plug-in for a method call with a single *String* parameter are less than the response times for a method call with a single *int* parameter, which are stated in Table 4.2, till the *String* length reaches up to 100. This is because all the XML data is serialized to characters in transmission and *int* data requires extra parsing of the characters representing an integer value while unmarshalling. In this experiment, length of 100 characters for a *String* has almost the same response time with an *int* that has the value 0.

Table 4.3: Results for *String* parameter (Single computer)

String length	CORBA	RMI	SOCKET
1 character	550	364	2055,7
10 characters	551,4	371,7	2055,8
100 characters	593,6	404,6	2180,7
1000 characters	734,3	462,4	2894,7
5000 characters	1140,3	859,2	5996,8
10000 characters	1348	1336	10222,3

Table 4.4: Results for *String* parameter (Multiple computers)

String Length	CORBA	RMI	SOCKET
1 character	399,8	290,6	397,3
10 characters	406,1	292,1	400,8
100 characters	443,6	335,9	424,2
1000 characters	798,2	663,7	611,7
5000 characters	1477,6	1298	940,5
10000 characters	2066,6	1734	2411,5

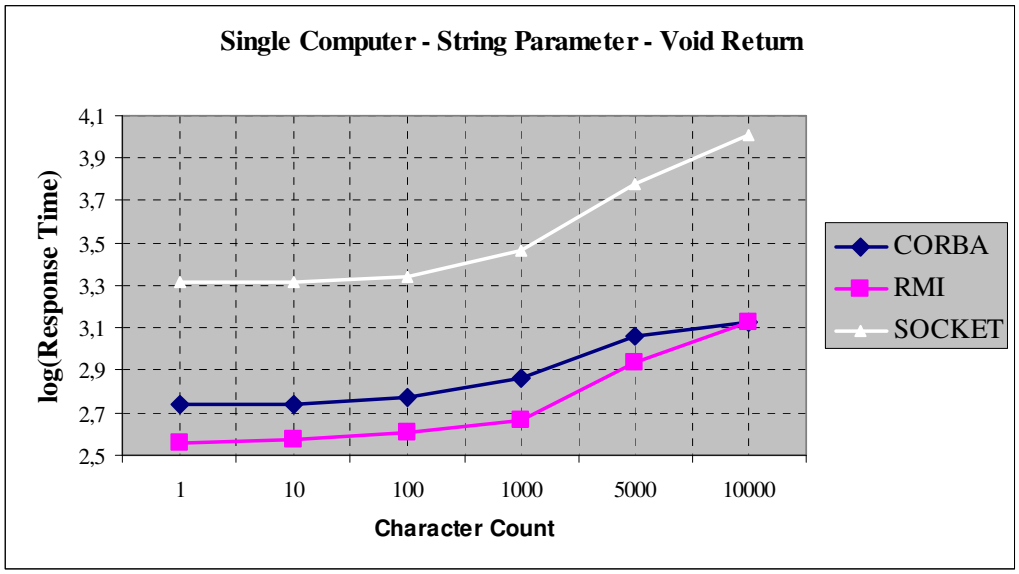


Figure 4.2: Results for *String* parameter (Single computer)

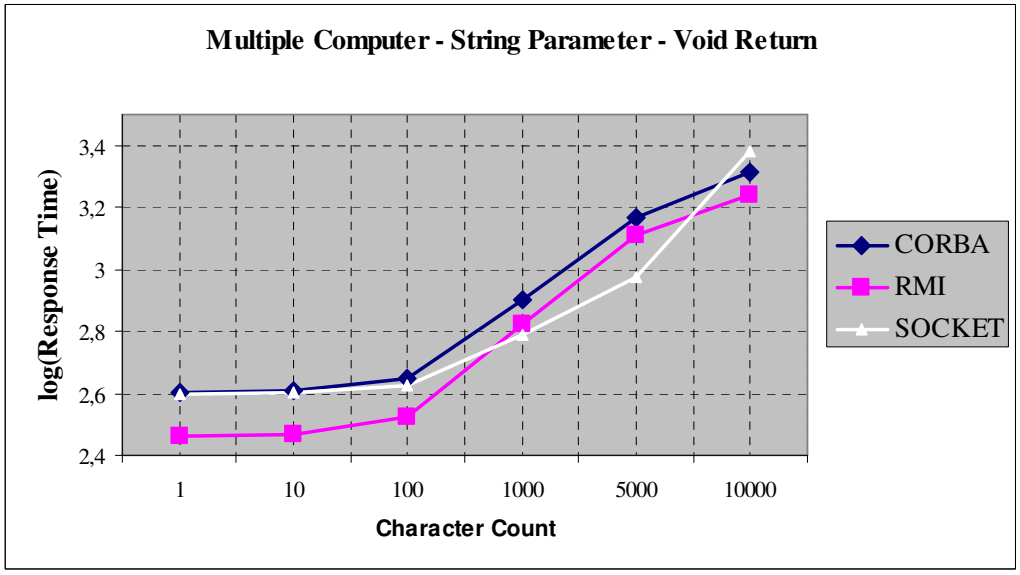


Figure 4.3: Results for *String* parameter (Multiple computers)

4.6. Call of Return Type Void with Primitive Type Parameter Array

In this experiment, a method call with a primitive type array parameter and void return type has been evaluated. For simplicity, *int* has been chosen for the primitive type with value of 0. In Table 4.5, Table 4.6, Figure 4.4.and Figure 4.5, the response time results have been stated for single computer and multiple computers run. The test cases are populated using the length of the *int* array. In addition to the general discussions stated in section 4.2, the following aspects regarding to the response times were identified:

(i) The response times of RMI plug-in are less than CORBA plug-in, which are also less than the Socket plug-in in both single computer and multiple computers run and in all the length variations.

(ii) The increase in the response times with the increase of array length in the Socket plug-in is much greater than other DOT plug-ins. This is because the increase in the size of the serialized object for the Socket plug-in is catastrophically compared with the other DOT plug-ins, since an array in an XML data means so many extra tags for each array object.

(iii) The response times of all DOT plug-ins for a method call with an *int* array parameter which includes only one primitive type inside are greater than the response times for a method call with a single *int* parameter, which are stated in Table 4.2, in both single computer and multiple computers run. This is because the serialization of primitive types requires less resource compared with the serialization of the Java objects.

Table 4.5: Results for *int* array parameter (Single computer)

Array Length	CORBA	RMI	SOCKET
1	531,1	501,4	2197,7
10	570,2	556,7	2572,7
100	681	605,1	4284,5
1000	973,1	773,3	28840
5000	1537,1	1438,6	261020,3
10000	2244,8	2164,3	530234,1

Table 4.6: Results for *int* array parameter (Multiple computers)

Array Length	CORBA	RMI	SOCKET
1	414,6	347	429,6
10	422,3	354,3	525,9
100	559,7	481	1283,2
1000	1262,9	1132,9	12784
5000	2818,3	2509,7	75660
10000	4810,7	4278	242175

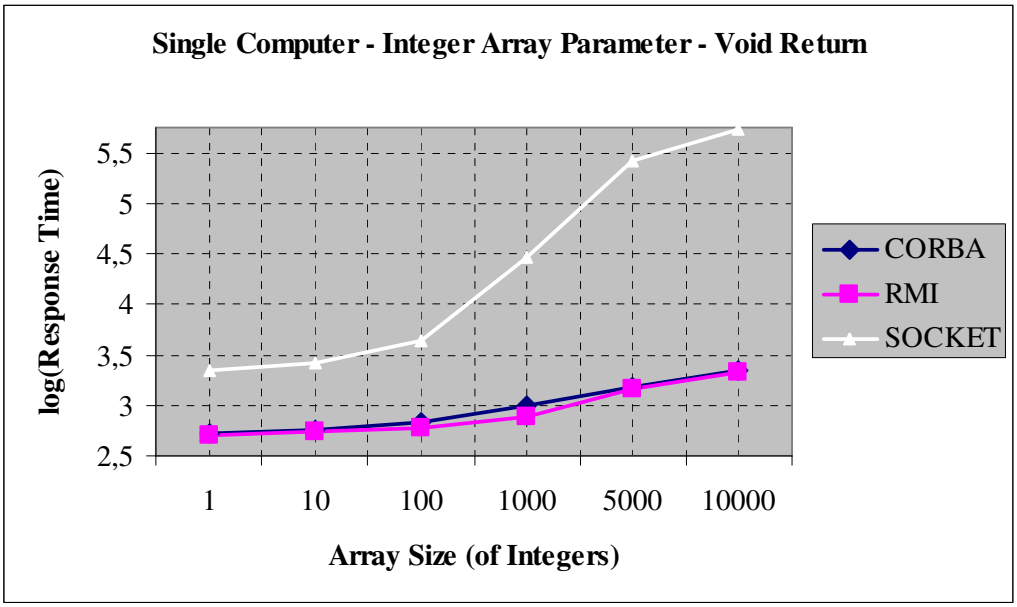


Figure 4.4: Results for *int* array parameter (Single computer)

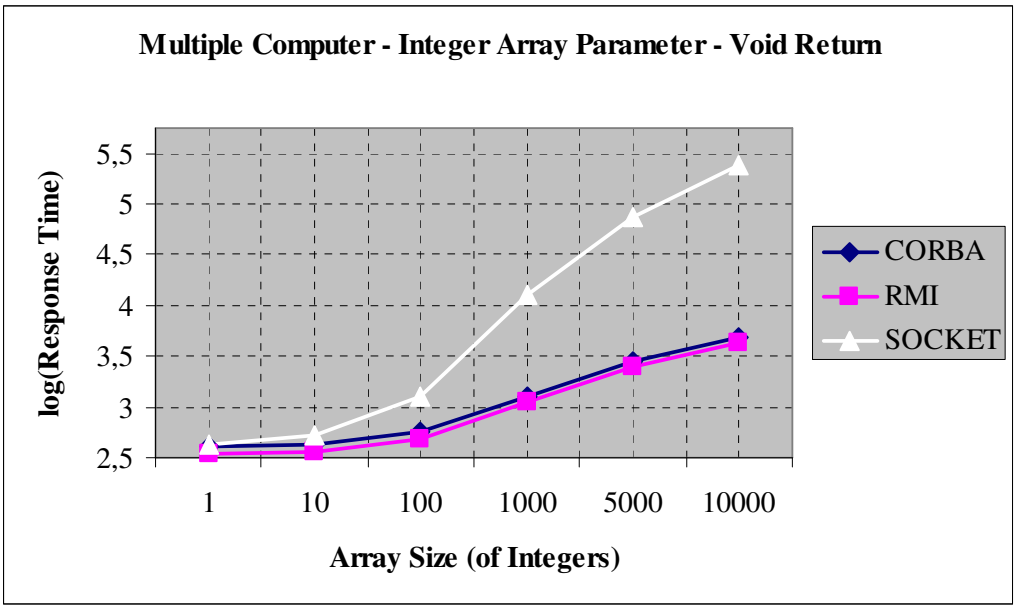


Figure 4.5: Results for *int* array parameter (Multiple computers)

In Figure 4.6, average response time results for primitive data type arrays with different lengths are stated. As compared with the results obtained in this study, it is identified from Figure 4.6 that CORBA has greater latency than RMI, which is consistent with the results in Table 4.5 and Table 4.6 that are obtained in this work.

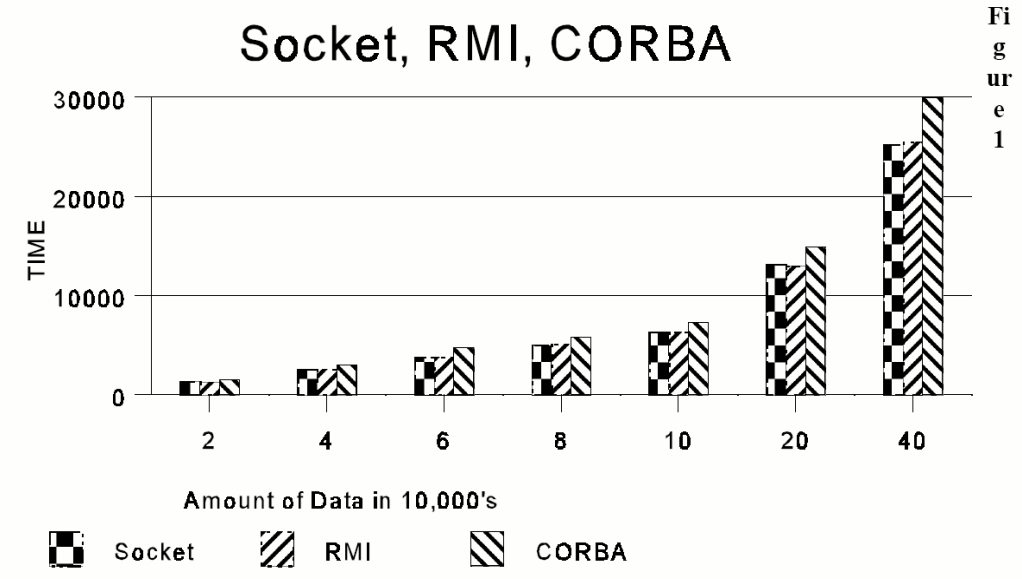


Figure 4.6 Results for int array parameter from [8]

4.7. Call of Return Type Void with Varying Length Parameter Array

In this experiment, a method call with an array of varying length type parameter and void return type has been evaluated. For simplicity, *String* has been chosen for the varying type with value of "a". In Table 4.7, Table 4.8, Figure 4.7.and Figure 4.8, the response time results have been stated for single computer and multiple computers run. The test cases are populated using the length of the *String* array. In addition to the general discussions stated in section 4.2, the following aspects regarding to the response times were identified:

(i) The response time of the Socket plug-in is much greater than other DOT plug-ins in both single computer run and multiple computers run, and in all the length variations.

(ii) In single computer run, when the length of the array is less than 100, the response times of CORBA plug-in are less than RMI plug-in; but when the length of the array is above 100, RMI plug-in has less response times compared with the CORBA plug-in.

(iii) In multiple computers run, the response times of RMI plug-in are less than CORBA plug-in in all the length variations.

(iv) The increase in the response times with the increase of array length in the Socket plug-in is much greater than other DOT plug-ins. This is because the increase in the size of the serialized object for the Socket plug-in is catastrophically compared with the other DOT plug-ins, since an array in an XML data means so many extra tags for each array object.

(v) The response times of all DOT plug-ins for a method call with an *int* array parameter are less than the response times for a method call with a *String* array parameter, which are stated in Table 4.5 and Table 4.6, in both single computer and multiple computers run. This is because the serialization of arrays of primitive types requires less resource compared with the serialization of arrays of complex Java objects.

(vi) The response times of all DOT plug-ins for a method call with a *String* array parameter which includes only one *String* inside are greater than the response times for a method call with a single *String* parameter, which are stated in Table 4.3 and Table 4.4, in both single computer and multiple computers run.

This is because the serialization of less complex Java objects requires less resource compared with more complex Java objects like arrays.

Table 4.7: Results for *String* array parameter (Single computer)

Array Length	CORBA	RMI	SOCKET
1	559,2	628	2071,5
10	596,8	646,7	2240,8
100	668,6	704,5	3668
1000	2299,3	1827,6	24928
5000	12658	5488,1	124890,2
10000	33654,6	9711,2	284030,5

Table 4.8: Results for *String* array parameter (Multiple computers)

Array Length	CORBA	RMI	SOCKET
1	416,7	366,5	436,3
10	437	402,6	531,9
100	664,4	647	1161,1
1000	2258,5	1809,7	9733
5000	9506,5	4310,7	60870
10000	21870,5	7370,4	120650,1

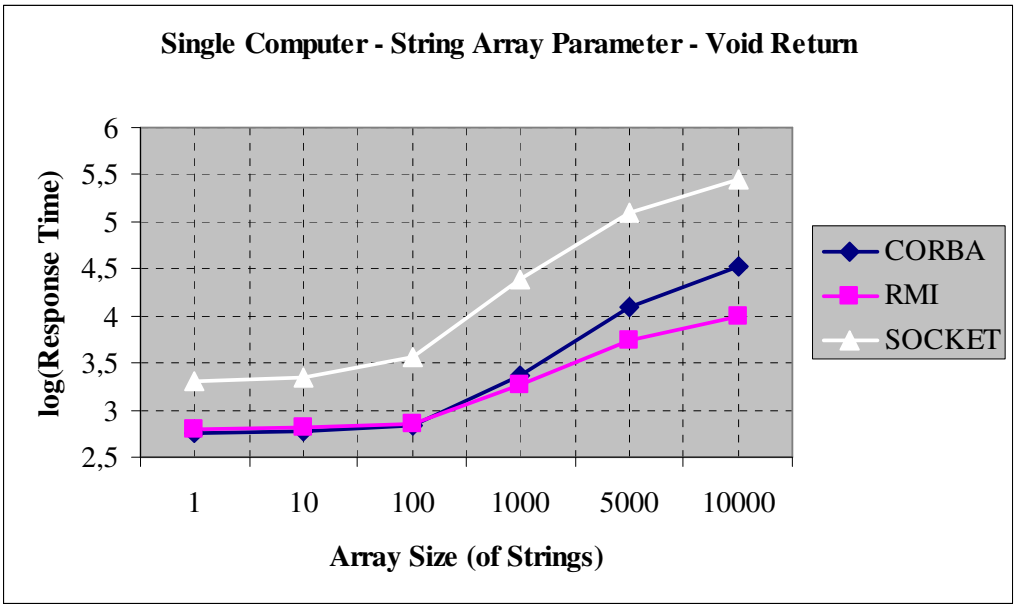


Figure 4.7: Results for *String* array parameter (Single computer)

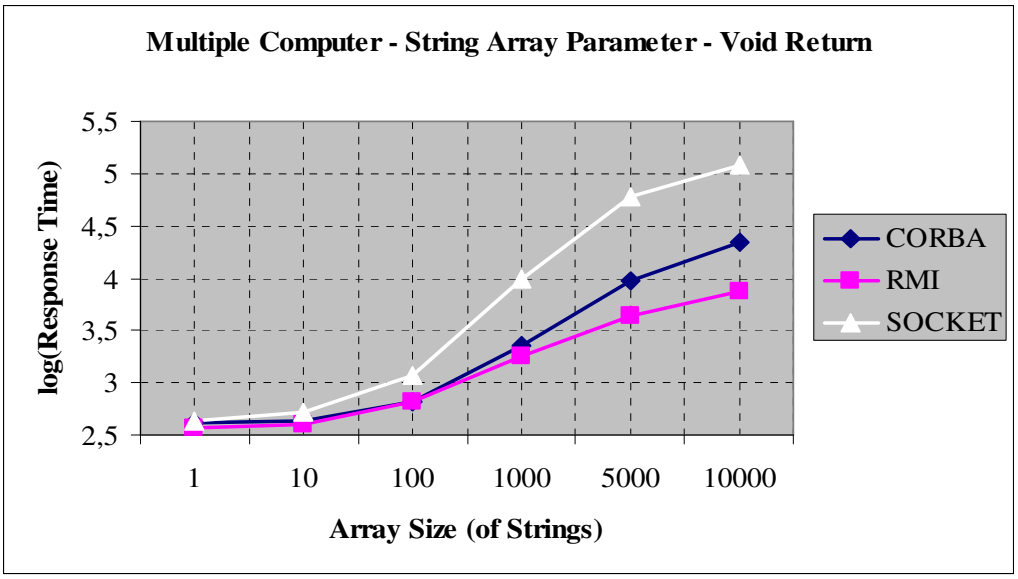


Figure 4.8: Results for *String* array parameter (Multiple computers)

In Figure 4.9, response time results for object array parameter are stated. In [5], Voyager has been chosen for the ORB vendor. As compared with the results obtained in this study, it is identified from the Figure 4.9 that CORBA has greater latency than RMI, which is consistent with the results in Table 4.7 and Table 4.8 that are obtained in this work.

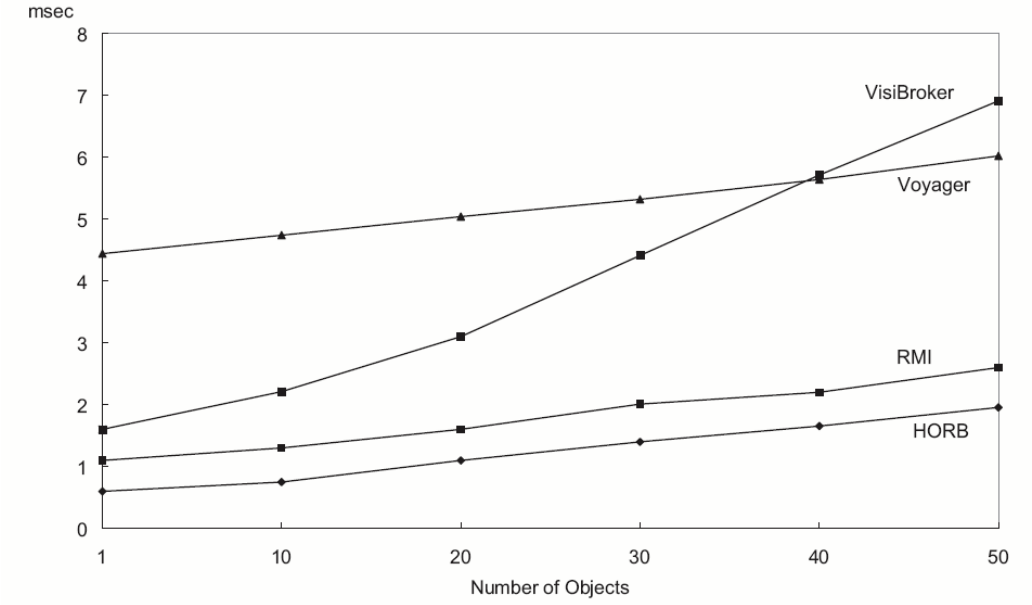


Figure 4.9 Results for object array parameter from [5]

4.8. Call with Return and Parameter Type of Primitive Type

In this experiment, a method call with a primitive type parameter and return type has been evaluated. For simplicity, *int* has been chosen for the primitive type with the value 0. In Table 4.9, the response time results have been stated, in which there is no column named SOCKET because in socket plug-in, method call is simulated by a hierarchical XML message sent through a socket that does not have the ability of returning a value of type something. In addition to the general discussions stated in section 4.2, it is identified that the response time of RMI plug-in is less than CORBA plug-in in both single computer and multiple computers run.

Table 4.9: Results for *int* parameter and return

	CORBA	RMI
Single Computer	504,6	321,8
Multiple Computers	409,3	282,7

4.9. Call with Return and Parameter Type of Varying Length

In this experiment, a method call with a varying length parameter and return type has been evaluated. For simplicity, *String* has been chosen for the varying length parameter type with value of "a" characters of varying length. In Table 4.10, Table 4.11, Figure 4.10.and Figure 4.11, the response time results have been stated for single computer and multiple computers run, in which there is no column named SOCKET because in socket plug-in, method call is simulated by a hierarchical XML message sent through a socket that does not have the ability of returning a value of type something. The test cases are populated using the length of the *String* data. In addition to the general discussions stated in section 4.2, the following aspects regarding to the response times were identified:

(i) The response time of RMI plug-in is less than CORBA plug-in in both single computer and multiple computers run and in all the length variations.

(ii) The response times of RMI and CORBA plug-ins for a method call with a single *String* parameter and return type are greater than the response times for a method call with a single *int* parameter and return type, which are stated

in Table 4.9, in both single computer and multiple computers run. This is because the serialization of primitive types requires less resource compared with the serialization of the Java objects.

Table 4.10: Results for *String* parameter and return (Single computer)

String Length	CORBA	RMI
1 character	556,1	409,4
10 characters	551,4	399,9
100 characters	595,2	426,4
1000 characters	731,2	556,1
5000 characters	1499,4	1323,2
10000 characters	1947,8	1935,1

Table 4.11: Results for *String* parameter and return (Multiple computers)

String Length	CORBA	RMI
1 character	434,2	316,2
10 characters	426,5	309,3
100 characters	503	390,5
1000 characters	1196,5	1080,9
5000 characters	2457,1	2396,1
10000 characters	3560	3457

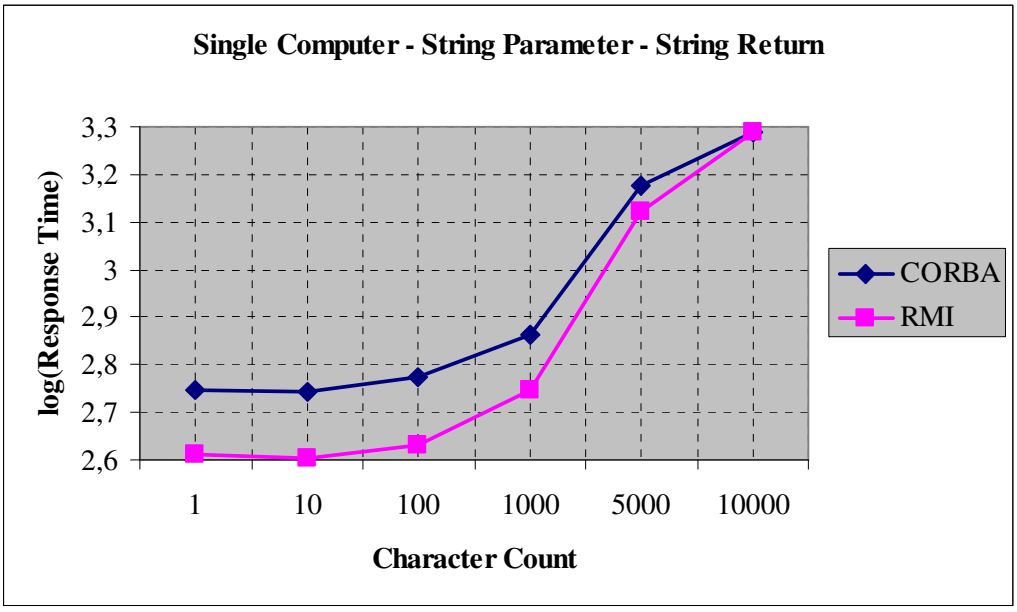


Figure 4.10: Results for *String* parameter and return (Single computer)

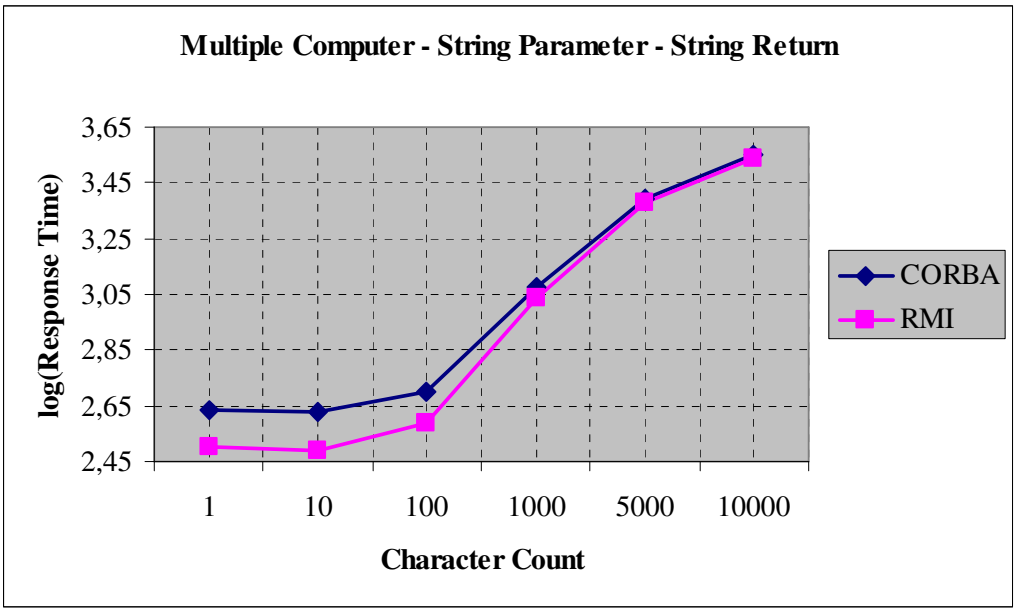


Figure 4.11: Results for *String* parameter and return (Multiple computers)

4.10. Call with Return and Parameter Type of Primitive Type Array

In this experiment, a method call with a primitive type array parameter and return type has been evaluated. For simplicity, *int* has been chosen for the primitive type with value of 0. In Table 4.12, Table 4.13, Figure 4.12.and Figure 4.13, the response time results have been stated for single computer and multiple computers run, in which there is no column named SOCKET because in socket plug-in, method call is simulated by a hierarchical XML message sent through a socket that does not have the ability of returning a value of type something. The test cases are populated using the length of the *int* array. In addition to the general discussions stated in section 4.2, the following aspects regarding to the response times were identified:

(i) In single computer run, the response times of RMI plug-in are greater than CORBA plug-in in all the length variations.

(ii) In multiple computers run, the response times of RMI plug-in are greater than CORBA plug-in till the length of the array reaches up to 1000. For the array lengths greater than 1000, CORBA has greater response times than RMI since delay for the large size of the serialized object over network becomes more significant.

(iii)The response times of RMI and CORBA plug-ins for a method call with an *int* array parameter and return type of length one are greater than the response times for a method call with a single *int* parameter and return type, which are stated in Table 4.9. This is because the serialization of primitive types requires less resource than the serialization of the Java objects.

Table 4.12: Results for *int* array parameter and return (Single computer)

Array Length	CORBA	RMI
1	532,2	754,5
10	580,9	756,1
100	693,7	774,8
1000	1102,8	1113,9
5000	2169,7	2300,9
10000	3517,8	3699

Table 4.13: Results for *int* array parameter and return (Multiple computers)

Array Length	CORBA	RMI
1	422,4	426,5
10	440,5	445,1
100	707,6	717
1000	2018,2	2026
5000	5214,2	4703,4
10000	9250,1	8250,5

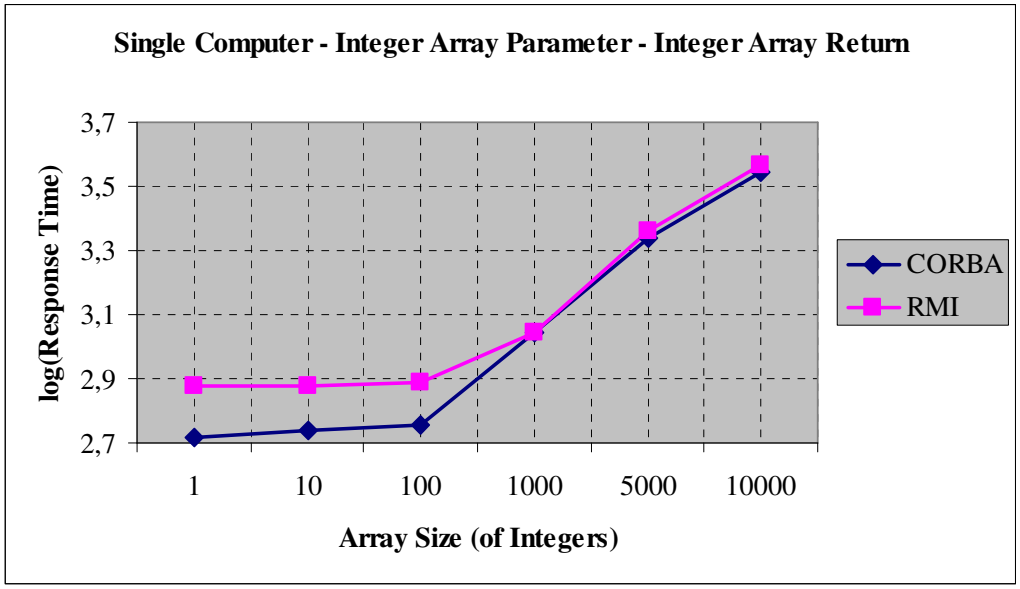


Figure 4.12: Results for *int* array parameter and return (Single computer)

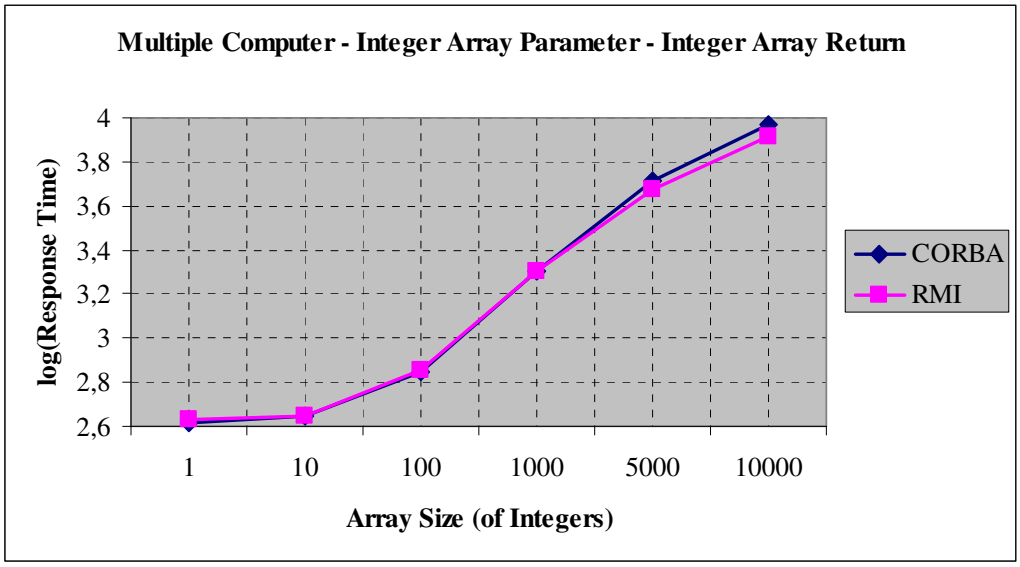


Figure 4.13: Results for *int* array parameter and return (Multiple computers)

4.11. Call with Return and Parameter Type of Varying Length Array

In this experiment, a method call with an array of varying length type parameter and return type has been evaluated. For simplicity, *String* has been chosen for the primitive type with value of "a". In Table 4.14, Table 4.15, Figure 4.14.and Figure 4.15, the response time results have been stated for single computer and multiple computers run, in which there is no column named SOCKET because in socket plug-in, method call is simulated by a hierarchical XML message sent through a socket that does not have the ability of returning a value of type something. The test cases are populated using the length of the *String* array. In addition to the general discussions stated in section 4.2, the following aspects regarding to the response times were identified:

(i) In both single computer and multiple computers run, the response times of RMI plug-in are greater than CORBA plug-in till the length of the array reaches up to 1000. For the array lengths greater than 1000, CORBA has greater response times than RMI since delay for the large size of the serialized object over network becomes more significant.

(ii) The response times of RMI and CORBA plug-ins for a method call with an *int* array parameter and return type are less than the response times for a

method call with a *String* array parameter and return type, which are stated in Table 4.12 and Table 4.13, in both single computer and multiple computers run. This is because the serialization of arrays of primitive types requires less resource compared with the serialization of arrays of complex Java objects.

(iii) The response times of RMI and CORBA plug-ins for a method call with a *String* array parameter and return type which includes only one *String* inside are greater than the response times for a method call with a single *String* parameter and return type, which are stated in Table 4.10 and Table 4.11, in both single computer and multiple computers run. This is because the serialization of less complex Java objects requires less resource compared with more complex Java objects like arrays.

Table 4.14: Results for *String* array parameter and return (Single computer)

Array Length	CORBA	RMI
1	559,7	795,1
10	607,6	826,4
100	904,4	1002,8
1000	3830,2	3221
5000	30888,6	10434,6
10000	76584,2	18719,7

Table 4.15: Results for *String* array parameter and return (Multiple computers)

Array Length	CORBA	RMI
1	435,3	513,8
10	484	552,9
100	899,9	1000,9
1000	4563	3043,4
5000	28850,3	9150,6
10000	71349,2	15444,1

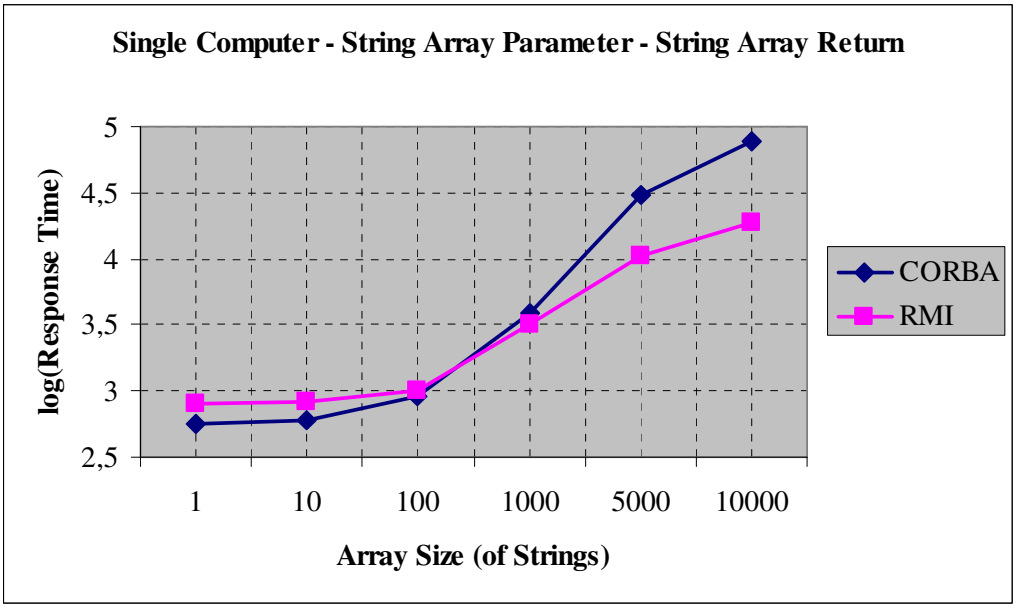


Figure 4.14: Results for *String* array parameter and return (Single computer)

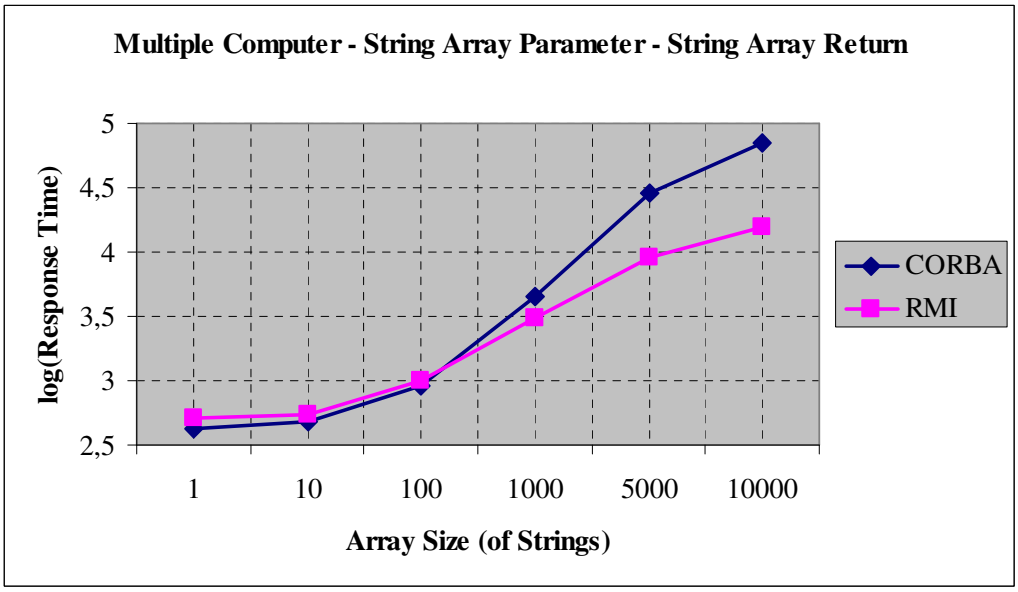


Figure 4.15: Results for *String* array parameter and return (Multiple computers)

CHAPTER 5

CONCLUSION

In this thesis, a way of design and implementation of a plug-in framework for distributed objects technologies (DOTs) has been presented. Three DOTs, namely CORBA, RMI and Java Sockets have been implemented in sample plug-in implementations. Using these sample plug-ins, a series of performance evaluations have been performed.

A plug-in framework has several advantages like run-time upgrading and increased extendibility. In this study, such a framework has been designed and implemented by using reflection and run-time class loading mechanisms of Java. It is possible to add this framework to any kind of application to distribute the tasks of that application using selected DOTs on a network domain.

Responding to different distributed architecture demands for the application domain leads the programmer to choose a suitable DOT that best satisfies these demands. To establish a baseline on the performance of these DOTs, a series of performance evaluations has been performed on the framework. During the evaluation part of this work, it has been observed that adopting a different DOT usage by an existing application using this framework is very quick and easy.

The series of performance evaluations revealed that using XML data on a Java socket as a DOT leads to catastrophically increasing response times with increasing data length compared to RMI or CORBA. Experiments also showed that response time increase is due to:

1. Marshalling mechanism of Castor libraries
2. The lack of ability of Castor libraries to understand the end of an XML message while unmarshalling incoming data from the socket.

Second problem is resolved by the socket plug-in through manually parsing the socket data before passing the data to the Castor libraries. To decrease the response times of this plug-in, as a future work, buffering mechanism on the socket which is used in parsing of the incoming message may be optimized. Also using another library rather than Castor may solve performance bottleneck due to marshalling mechanism.

REFERENCES

[1] Pilone, M., *Plug-ins & Java,* Dr. Dobb's Portal (http://www.ddj.com), 2004

[2] Object Management Group, http://www.omg.org/, Last access date: September 2006

[3] Sun Microsystems Inc., http://www.sun.com/Java, Last access date: September 2006

[4] Ahuja, S. P., QuinTao, R., *Performance Evaluation of Java RMI: A Distributed Object Architecture for Internet Based Applications*, 2000

[5] Hirano, S., Yasu, Y., Igarashi, H., *Performance Evaluation of Popular Distributed Object Technologies for Java*, 1998

[6] Munoz, C., Zalewski, J., *Architecture and Performance of Java-Based Distributed Object Models: CORBA vs RMI*, 2001

[7] Juric, M. B., Rozman, I., Hericko, M., *Performance Comparison of CORBA and RMI*, 2000

[8] Eggen, R., Eggen, M., *Efficiency of Distributed Parallel Processing using Java RMI, Sockets and CORBA*, 2001

[9] Szyperski, C., *Component software: Beyond object oriented programming*, Addison-Wesley Pub Co, 1997

[10] Chatley, R., Eisenbach, S., Magee, J., *Painless Plugins*, 2003

[11] Handschuh, S., *OntoPlugins – A Flexible Component Framework*, 2001

[12] Object Technology International, Inc. *Eclipse Platform Technical Overview*, Technical Report, IBM, July 2001

[13] Oriezy, P., Medvidovic, N., Taylor, R., *Architecture-Based Runtime Software Evolution*, In ICSE '98, 1998.

[14] Liang, S., Bracha, G., *Dynamic Class Loading in the Java Virtual Machine*, 1998

[15] Chatley, R., Eisenbach, S., Kramer, J., Predictable *Dynamic Plugin Systems*, 2004

[16] Magee, J., Dulay, N., Eisenbach, S., Kramer, J., *Specifying Distributed Software Architectures*, In Proceedings of the 5th European Conference on Software Engineering, Sitges, Spain, 1995, pages 137–154.

[17] Magee, J., Kramer, J., *Concurrency – State Models and Java Programs*, John Wiley & Sons, 1999.

[18] Couloris, G., *Distributed Systems, Concepts and Design*, Addison-Wesley Publishing, 1994

[19] Oberleitner, J., Gschwind, T., Jazayeri, M., *The Vienna Component Framework Enabling Composition Across Component Models*, In Proceedings of the 25th International Conference on Software Engineering (ICSE'03), 2003

[20] World Wide Web Consortium, http://www.w3.org, Last access date: September 2006

[21] The Castor Project, http://www.castor.org, Last access date: September 2006

[22] JacORB, http://www.jacorb.org, Last access date: September 2006

[23] Hall, R., *A Policy Driven Class Loader to Support Deployment in Extensible Frameworks*, 2004