Comparative Evaluation of Object Request Broker Technologies

By

Fadi Louis Nammour

Submitted in partial fulfillment of the requirements
for the Degree of Master of Science

Thesis Advisor: Dr. Nasha't Mansour
Department of Computer Science
LEBANESE AMERICAN UNIVERSITY
November 2002
I grant to the LEBANESE AMERICAN UNIVERSITY the right to use this work, irrespective of any copyright, for the University's own purpose without cost to the University or to its students, agents and employees. I further agree that the university may reproduce and provide single copies of the work, in any format other than in or from microforms, to the public for the cost of reproduction.
Lebanese American University

Graduate Studies

We hereby approve the thesis of

Fadi Louis Nammour

Candidate for the Master of Science degree*.

(signed) ____________________________
(chair) ____________________________

date  Jan. 29, 2003

*We also certify that written approval has been obtained for any proprietary material contained therein.
Comparative Empirical Evaluation of Object Request Broker Technologies

Abstract

By

Fadi Louis Nammour

CORBA, Java RMI, and WebServices are among the most popular Object Request Broker technologies. These technologies have allowed replacing message-based network collaboration with method invocations through remote object calls over network clusters. In this thesis, we propose several criteria for evaluating and comparing distributed object computing technologies. Empirical criteria include performance, memory management, scalability, load balancing, fault-tolerance, fail-over and clustering. Qualitative criteria include maintainability, testability, repeatability, ease of learning, and portability. We use these criteria to empirically and qualitatively evaluate and compare CORBA, Java RMI, and WebServices. The results of the study indicate that no technology can be used as a perfect solution to all problems in distributed object software. For example, CORBA is superior when load balancing is used, while Java RMI is the best technology to use when rapid deployment and development are required, while WebServices are the best solution for scalability and portability.

Keywords: Distributed object computing, RMI, CORBA, Web Services, Software engineering
To my parents
ACKNOWLEDGMENTS

I would like to thank my advisor Dr. Nasha’t Mansour for his guidance throughout my M.S. studies. Thanks are also due to Dr. May Abboud and Dr. Ramzi Haraty for being on my Thesis committee.

I would like to express my sincere gratitude to the Lebanese American University whose technical and educational support during my graduate studies made it all possible.

Finally, I would like to thank my friends and family for their long support.
# Table of Contents

1. Introduction .................................................................................................................. 10
   1.1 Literature Review .................................................................................................. 13
   1.2 Brief Discussion of the ORB Technologies Used ..................................................... 15
      1.2.1 Java RMI ........................................................................................................ 16
      1.2.2 CORBA .......................................................................................................... 16
      1.2.3 WebServices .................................................................................................... 17
      1.2.4 DCOM ............................................................................................................ 17
   1.3 Scope and Objectives ............................................................................................... 19
   1.4 Organization ............................................................................................................ 21

2. RMI, DCOM, CORBA, WebServices ............................................................................ 22
   2.1 Java RMI ................................................................................................................. 24
   2.2 DCOM ..................................................................................................................... 28
   2.3 CORBA .................................................................................................................... 31
   2.4 WebServices .......................................................................................................... 33

3. Proposed Evaluation Approach .................................................................................... 37
   3.1 Overview ................................................................................................................ 38
   3.2 Enterprise Application ............................................................................................ 40
      3.2.1 The Idea .......................................................................................................... 40
      3.2.2 The History ...................................................................................................... 41
      3.2.3 The Technology ............................................................................................... 41
      3.2.4 The Design ....................................................................................................... 42
      3.2.5 The Class-Diagram .......................................................................................... 45
   3.3 Evaluation Criteria .................................................................................................. 59
      3.3.1 Quantitative Criteria ......................................................................................... 61
      3.3.2 Qualitative Criteria .......................................................................................... 78

4. Experimental Results .................................................................................................. 91
   4.1 Setup / Design ....................................................................................................... 92
      4.1.1 Operating System Installation ......................................................................... 93
      4.1.2 Networkability ................................................................................................. 93
4.1.3 Software Installation ................................................................. 93
4.1.4 Component Configuration ......................................................... 94
4.2 Results ...................................................................................... 96
4.2.1 Empirical Results ................................................................. 96
4.2.2 Qualitative Results ............................................................... 97
4.3 Discussion ............................................................................... 101
4.3.1 General Evaluation .............................................................. 101
4.3.2 Intra-Criteria Effects ............................................................. 103
5. Conclusion and Future Work ...................................................... 105
References ...................................................................................... 106
List of Figures

Figure 1: Application Components ................................................................. 40
List of Tables

Table 1: Evaluation criteria of ORB technologies .............................................................. 59
Table 2: Indexes of the experimental criteria ................................................................. 96
1. Introduction

As the revolutionary idea of *Object Oriented programming* found its way into programming languages, and as networking support in such languages became a standard, many software vendors realized the opportunities that lie within the distributed world. With the proper network configuration and the software to control the machines, this relatively cheap “virtual machine” offers enormous processing power. Distributed object computing is a recent trend in computing that aims to exploit the computing power of this “virtual machine”.

Not every software product that utilizes network resources belongs to the group of distributed object software. The networked-machines paradigm was there long before object-oriented computing existed. RPC-based Unix operating systems were playing the major part in the computer industry and were replacing the mainframe soon before object-oriented programming languages (notably C++) made their entry. But it is a well-known fact that these object-oriented programming languages had all the tools needed to start the distributed object architecture.

Three categories of distributed object software can be identified:

- **Object-oriented software collaborating over the network.** Any object-oriented application that makes calls to the network libraries for communication purposes can be listed in this category. Such an application might be partitioned over a set of machines, the parts communicating over the network performing either the same task (parallel programming) or complementary tasks (distributed application). However, the parts – each a set of objects – should have a transitive closure structure. This means that the logical connection between objects on one machine is stronger than the connection with other objects on remote machines. It can be easily proven that object-oriented applications running within the parallel programming framework obey this characteristic. Since all machines are performing the same task, they must all have copies of the objects needed to fulfill this task.

- **Fully object-oriented software utilizing networked resources.** An application belonging to this category might have access to a resource (such as memory or
processor) located over the network. The application would use the resource as if it were located on the same machine that it is running on. The application would migrate the code and data to a remote machine, perform specific tasks on that machine, and return the result to the application for further processing. Thus the remote resource is peripheral and not an integrated part of the application resources.

- Object-based software with computing power resulting from distributing the objects over the network. As opposed to the first category, the local connection between the objects does not have to be stronger than that between remote objects. The distribution might or might not be even, and the objects that perform the most work might all be on the same machine or otherwise located at different nodes. In such architecture the resulting collaboration of all the machines is what leads to the success or failure of the computation, and no resource is peripheral unless explicitly declared (a hybrid of this and the second category can be desirable sometimes).

The following definition of distribution is adopted throughout this thesis:
Distribution is the process of taking a complete solution (or product) and dividing it into parts that can be placed on connected nodes [OrHa98, Bai00]. Here the complete solution includes the code and other parts like application data, software users, administration policies, hardware or operating system resources, and probably other software that need to be worked with.

Distributed software can be defined as: Distributed software is any software that can be “successfully” distributed over a number of nodes without compromising performance, resource management, or users’ interaction with the system [Öbe99, Zah00].

Distributed object software possesses the following properties:
1. Completely Object-Oriented: the software should consist of objects collaborating to achieve a solution of perform a computation
2. **Partitioned over Connected Nodes:** the partitions or parts of the software should consist of at least objects making up the software. The partitions could include data and other resources as well.

3. **The Object communication is Delegated:** the delegation can be orchestrated by other software acting as a middleware, or through the operating system.

4. **The Class Structure is Collapsible:** it should be possible for the system in its class design to collapse or reorganize to run on a single node or processor.

The above definition clearly respects the categorization of distributed software mentioned earlier, and that is seen clearly in the second property. The third property makes it clearer by delegating object communication. This means that “downloading” objects does not qualify software as distributed. Many applications (applets in the Java lingua) download the classes and instantiate objects on the local node (machine). This clearly does not make them distributed.
1.1 Literature Review

Most of the previous work on Distributed Object Computing technologies discusses each technology independently. Not much work has been done for comparing these technologies in a comprehensive way. Orfali and Harkey [OrHa98] discusses the performance of CORBA and compares it with others. It concludes that as far as performance is concerned, CORBA seems very competitive to DCOM, whereas network technologies like CGI fail dramatically in almost every aspect studied. The main course of action taken by Orfali is to build a small application with a server and a client component, and then run the application in a networked environment, and study the performance based on the time needed for the client to send the request and get back the results. Oberg [Öbe99] focuses on building distributed applications that use RMI to perform their services, and it extends the topic to discuss dynamic class-loading and object migration. In addition to object RMI, [Öbe99] explains EJB components – Enterprise Java Beans – and the ease through which they can be built. RMI plays an important role in the J2EE framework.

An interesting study is that conducted by Rosen in [RoCu98]. In this survey CORBA is used as a network transportation mechanism, while COM is used as the component programming model, and both are used concurrently to build distributed applications.

In his book [Pope98], Pope tackles the ideas of distributed objects through a comprehensive and general reference guide. Pope tries to replace the traditional programming approach to new technologies through explaining in a qualitative way the features of CORBA. CORBA is a standards-based technology with implementations in many different programming languages, this book forms a good reference for people who are interested in pure CORBA rather than an introduction to CORBA and how to get two machines to talk. CORBA's concepts are explained in a scientific way through a necessity-availability approach. CORBA is introduced as the answer to many problems that arise in distributed object software. [Pope98] avoids to go into the comparison game, and limits the book mostly to CORBA. We believe that such a book would play the judge when attempting to expand our study to that of comparing CORBA implementations. As
we point out later in this study, CORBA itself has different third-party implementations, and the Java-CORBA interfaces allow implementations to be used at run-time rather than attached directly to the code. For this reason, the same application can be used to study different CORBA implementations without modifying the code.

Wilkinson and Allen [WilA99] discusses several procedures for achieving parallel behavior in network applications. With the wealth of network protocols that enable object collaboration, it would be interesting to consider combining network resource into one big virtual machine. The many packages that implement the MPI – or the Message Passing Interface – make programming in such a model a similar experience to that of distributed programming. The contrast, however, enables us to better understand the structure of both models. We used some of the ideas discussed in [WilA99] to enable our application with load-balancing features. We replaced the MPI model with that of messaging through method calls. While many MPI implementations use direct TCP/IP, distributed technologies build their marshalling model on top of it. The general idea of a single master and multiple slaves still holds in our implementation, but we were able to add dynamic cluster growth while still maintaining acceptable load balancing. We consider [WilA99] as the best reference with dealing with such programming models.
1.2 Brief Discussion of the ORB Technologies Used

Among the many ORB technologies currently available for building distributed object software, the ones we will be mostly interested in are Java RMI, CORBA, DCOM and WebServices. We chose these technologies and not others for the following reasons:

1. Java is a fully object-oriented programming language. All of the four technologies are either fully implemented in Java or have a Java extension of some sort. We feel that Java is the standard for enterprise software development and will surely continue to grow as the best programming language for the enterprise.

2. Three of the four technologies are platform independent, with DCOM being limited to the Windows operating system. The only reason that made us study DCOM along with the other technologies is that DCOM has gained significant success over the past couple of years, and is now an intrinsic part of all Microsoft server software. We do not know of any DCOM implementation for other operating systems, and we do not feel that such implementations might emerge one day. Being part of a proprietary solution, and the fact that its interfaces are not available for the open-source community, and the current involvement of Microsoft in it. Net framework, all of these reasons stand in the way of any DCOM implementations away from the Microsoft Windows operating systems.

3. Three of the four technologies come as part of a standard product. While Java RMI and CORBA come as part of any Java Development Kit, DCOM comes pre-installed and configured on almost all Microsoft operating systems. Notably Windows 2000 and newer operating systems have DCOM as a pre-configured service that many other services use. And it is very easy to build Java applications that use DCOM. Microsoft has the SDK for Java version 4 available for free download. This SDK is only required when building the application, but once built the application can run on any Windows platform. As for WebServices, the Java implementation that we used requires a web server with a Servlet implementation. The one we used is part of the Jakarta project, the famous Tomcat.
The reasons mentioned above make the four technologies primary candidates for developing distributed object software. The increasing number of developers using these technologies makes it easy to look for answers when problems arise. The literature is also rich with books and papers discussing one or more of the four technologies. Almost all books on Java discuss RMI in one or more chapters, and books on CORBA and DCOM occupy several shelves in almost all computer science libraries. There is a wealth of publications discussing the four ORB technologies, and with the growing hype of the enterprise, WebServices have occupied a front-row in the enterprise software industry.

Now that we have clearly decided which ORB technologies to include in our thesis, let us explain them briefly and see how they achieve the goal of object communication over a network.

1.2.1 Java RMI

Java RMI, or simply RMI, stands for Remote Method Invocation. The technology resulted from the need to replace the aging RPC – Remote Procedure Call – network technology. Java RMI is a pure Java technology, and its implementation is restricted to the Java platform. This inherent connection between RMI and Java makes RMI the best choice for all-Java applications. While other technologies might require third-party software or plug-ins to work, this technology is built into the virtual machine – the JVM.

RMI employs the same concept that most ORB technologies use. Objects on different machines communicate through a naming server, and every object has a unique identifier throughout the entire network. The difference however is in the mechanism through which these objects are fetched, operated on, and disposed of. RMI has a unique all-Java approach when it comes to object manipulation. Since object manipulation is performed within the JVM, no overhead is incurred, making RMI very efficient in that context.

1.2.2 CORBA

CORBA, or the Common Object Request Broker Architecture, is a very reliable ORB technology because it is based on a standard protocol that has evolved in time, benefiting from the improvements in both hardware and operating system technologies.
CORBA has been in the ORB scene for a long time now. Even before the Java programming language, CORBA was driving enterprise applications, enhancing collaboration between applications written in different programming languages. As a matter of fact, it is this feature that made CORBA the primary choice for all distributed applications.

But CORBA imposes a new language that application developers must learn to get their work done. In order to unify the object communication process, CORBA introduced IDL, a new and revolutionary language, replacing message-based object communication to call-based object handling. And regardless of the programming language used, object architectures written in IDL immediately translated into interfaces and components that can be directly accessed from with the application in its native language. The concept of proxies and skeletons was thus introduced, allowing programs to operate on remote objects as if they were local.

1.2.3 WebServices

The challenging and critical mission was that of allowing object communication to grow beyond the intranet to the outreaches of the “Wide Area Network” – the network of all networks – the Internet. Applications started growing and we needed a way to allow them to communicate, even through the rough textures of the modern firewall-driven Internet. WebServices provided the much-needed bridge between object communication at the intranet level, and remote method invocation at the Internet level. CGI was doing the job of “posting” information from a client to a server over the commonly used HTTP protocol. With the birth of WebServices, a web-server now hosts the remote objects, and these objects are activated and deactivated through marshalling calls, a procedure identical to that used by RMI and CORBA. But every new luxury comes at a price, and the price that developers and enterprise applications alike had to pay was performance.

1.2.4 DCOM

Microsoft had something in the works, something that was inline with what every third-party vendor had in mind. The Windows operating system hosted most of the enterprise applications that required object communication, so it was only natural for
Microsoft to come-up with its own ORB technology. And that it surely did. DCOM – Distributed Component Object Model – came as a “distributed” version of the notorious COM, the driving force behind all Microsoft products, and the primary object technology making-up the line of operating systems ever since the first version of Windows95 was released. And even as Microsoft itself is trying to get rid of this ORB technology that it bound developers to for so long, it cannot escape the fact that DCOM will stay in duty for a very long period of time. The amount of applications that rely on DCOM to deliver object communication to the enterprise is very huge, and replacing it with new technology would seem a little too much for most developers.

DCOM allows objects to communicate by having them registered in the operating system. With every object possessing a unique identifier, the operating system can load the library files associated with every object identifier, and allow applications to target the object through a marshalling process similar to that of CORBA and RMI. Therefore applications using remote objects would do so as if the objects where COM objects registered at the local machine.
1.3 Scope and Objectives

Previous work have mostly considered each technology independently. Where comparative studies were made, they had limited scope and were not based on comprehensive criteria. In this thesis, we present a broad framework that allows us to identify a variety of characteristics of the ORB technologies. These criteria are (a) performance, (b) scalability, (c) memory management, (d) load balancing, (e) fault tolerance, (f) fail over, (g) clustering, (h) maintainability, (i) testability, (j) repeatability, (k) ease of learning, and (l) portability. The primary objective is to understand, through comparative evaluation, the behavior of the three ORB technologies in a distributed software environment.

We develop a distributed object application that we use to study the ORB technologies. We focus on the distributed nature of the application rather than on the services it delivers. This application is used to evaluate and compare CORBA, Java RMI, and Web Services. Our empirical and qualitative results show that each technology behaves differently under different scenarios and that each has strengths and weaknesses. For example, CORBA is superior when load balancing is used, while Java RMI is the best technology to use when rapid deployment and development are required, while WebServices are the best solution for scalability and portability.

These results can be useful to DOC software developers in selecting technologies that are adequate to their applications.

Our comparative evaluation relies on certain criteria that are common to all ORB technologies. Applications built for use in a distributed environment are expected to be far more reliable than their single-node counterparts. The success of the software is determined by two factors. First there is the distributed technology being used to couple the various parts of this application, and second there is the ability of the development team to produce software that best utilizes this technology to solve the various problems. When selecting the evaluation criteria against which we will test the ORB technologies, we will try to select those that reflect the challenges that face distributed software and the requirements that, if not met, will undermine the entire effort. However, our study is limited in both time and resources, so we are forced to limit the scope of this study to
only those criteria that are both very common and at the same time critical in the distributed development process.

We finally grade the technologies according to the criteria studied. Every technology will have an evaluation grade that we proposed after having studied it and evaluated it subject to the respective criteria. We feel that such a grading system will be of value to those interested in using these technologies to build distributed software.
1.4 Organization

The rest of the thesis is organized as follows. Chapter 2 further explains the DOC technologies used. Chapter 3 presents our proposed criteria for evaluating the DOC technologies. Chapter 4 presents the experimental setup and lists the results of our work and discusses them. Chapter 5 presents a conclusion of our work and what can be done in future work.
2. RMI, DCOM, CORBA, WebServices

Distributed Object Computing can only be achieved if a mechanism is used that will provide object-to-object communication over a network. This mechanism is usually available as part of either the operating system such as the case of DCOM under the Windows operating system and RPC under most UNIX implementations, as part of the programming language such as Java RMI, or as part of an independent vendor solution such as CORBA and WebServices. Whatever the source or location of this mechanism, it has to allow for object communication. A general term has been agreed upon to call this mechanism. Object Request Broker is any library, module, or service that facilitates object communication over a network. It is important here to state a few distinguishing factors about ORB technologies:

- There is no standard that all ORB’s follow. Some ORB’s (like CORBA and WebServices) have their own standards, and vendors should comply with these ORB specifications. But when it comes to the general concept of object communication, there does not exist any unified procedure or standard for providing object services over a network. Although CORBA comes as a result of a global effort to unify the way objects communicate, it failed to gain the approval of Microsoft, the makers of DCOM, who chose to take on a new technology that matches CORBA. WebServices, however, came late to unify all major software vendors under one standard for object communication. It can be said that the WebServices framework is the only ORB that enjoys global approval.

- The lack of a generally approved standard has lead to a state of chaos in the distributed software technology. Many vendors created their own ORB’s, introducing them as part of an operating system as in the case of Microsoft’s DCOM, or bundling and promoting them with the programming language as in the case of Java RMI and CORBA. Many ORB’s work only on certain operating systems, require specific programming language support, or even rely on specific network protocols not generally accepted. The chaos grew as major software solutions relied on such ORB’s, and the success of the application(s) promoted the underlying ORB to a de-facto standard the hard way. As an example, all
Microsoft products rely on COM, which has DCOM as its natural network extension. This promoted DCOM to a standard in the Microsoft world, and many software vendors found themselves using DCOM to promote their software under the Microsoft operating systems.

- At a technical level, the growth of the Internet brought a new challenge to the various intranets using distributed software solutions. Security issues jeopardized the ability of distribution, and the fact that all ORB technologies prior to WebServices use a predefined port to conduct the object communication meant that hackers could now use these ports to launch attacks on the network and probably gain access to otherwise secure resources. And installing a firewall to trap hackers out of the network very soon resulted in trapping the software inside the network, and when the business boundaries start to expand the software could no longer keep up with it. While some ORBs such as RMI forbid anonymous “objects” access to the software resources, many other ignore the security issue and rely on the applications in this context.

In this chapter we will attempt to discuss four ORB technologies that are of considerable importance when building distributed object software. These technologies have played important roles in promoting distributed software throughout the early stages of networked enterprise application development. As these technologies grew, their importance became clear as enterprise frameworks for building modern applications relied on such technologies to deliver the performance and extendibility that was needed in the enterprise. For example, the J2EE standards for enterprise applications in Java have CORBA and Java RMI at the core of the implementation, using them to connect application components in any network environment. Another example is WebServices.

At the time of writing this thesis, it is clear that web application – and to that extent enterprise application – are starting to get ported from the network model to the webservices model, where application components are placed at the locations that best host them, and this is making webservices grow in applications to the extent that the n-tier application model has webservices at its core. Even Microsoft, which has promoted DCOM for years, is now promoting XML webservices through its new .Net framework.
We will discuss every one of these technologies in a sub-section, describing what it does and how it does it.

2.1 Java RMI

In an effort to promote its cross-platform programming language, Sun Technologies decided to enable object communication solely within the framework of the Java programming language. The challenges that such a decision imposed on the development process soon became obvious. Java is not only a cross-platform programming language, but also a cross-compiler language as well. By cross-compiler we mean that any compiler, whether built by Sun or any other third party software vendor, should take in the same source code and generate the same compiled byte-code, as it is known in the Java circles. Such byte-code or compiled class is interpreted and run by specially developed runtime environment applications, notably the JVM.

Implementing an all-Java object communication model means that an object – in its byte-code form – should be able to delegate communication to another Java object, which in turn should perform the network operations required to communicate and forward the method invocations through the JVM. The classes of the application are called the stubs, which are specially generated classes that expose the object’s functionality and methods to the outside – RMI – world. Let us go through the steps in some detail to better understand this operation.

Any class that is written in Java can be exposed as an RMI service class. To do that, the class should either extend UnicastRemoteObject – a special class whose job is to expose the methods to the RMI registry, or implement java.rmi.Remote – an interface whose methods need be implemented in the class itself. Whichever of the two methods is followed, the methods that are exposed should declare the exception RMIException to be thrown. We found that this is a technical constraint more than a logical one, since the RMI compiler – a special program that reads the byte code and generates the stubs – need to know which methods to export, and throwing such an exception seems to be a marker for these methods.

After properly labeling the class, and after running the RMI compiler against the classes that we intend to use in the distribution process, the server part of the application
is now ready. A special instance of the RMI registry need be run at the node where the server objects are hosted, since communication is performed through a special naming server that resolves abstract and generic names into RMI-specific object locators.

The RMI packages included as part of the Java language allow client applications to create proxy or stub instances that have the same method names and signatures as the server ones, provided that the RMI locator is known for these objects. The RMI framework that resolves the names into object instances, forwards the method parameters – or marshals them in distributed terms – to the server where the object is located. The RMI registry at that server node then translates the invocations to locale ones. The result of the local invocation is then marshaled back to the calling naming server, which completes the cycle by forwarding these results to the local client object. This mechanism is very much a standard in object communication, and it will be encountered whenever object communication is implemented.

There are large numbers of articles that present RMI application development techniques and explain how to build applications that use RMI efficiently. In this section we do not intend to add another technical introduction to RMI programming, however we will discuss the inner workings of RMI, and we feel that of the many ORB technologies that we consider in this thesis, RMI is the most open in its architecture. The RMI implementation code is available and can be read and analyzed to further understand RMI. In addition, the RMI compiler that takes in the compiled Java byte-code representing the remote class itself generates both the Java source code and the compiled byte-code for the proxy or stub classes. By looking at this proxy code, we can learn a lot about the way RMI works, and it is this knowledge that we are going to present in this section.

We will use a class in our application to point out the inner workings of RMI. The class is called \texttt{NamingServerImpl}, and from its name we an tell that it is our implementation of a naming server, a mediator class that allows us to implement certain features such as fail-over and clustering. We will look at only one method of this class, simple the \texttt{getURL()} method, which takes in a \texttt{string} and returns a modified \texttt{string} object. The following is the method written in Java. We will not discuss or explain its syntax, and following it immediately is the RMI-generated proxy method:
public String getURL(String name) throws RemoteException
{
    String url = "rmi://" + (String)mappings.get(name) + ":" +
    Util.getPropertyInt("RMI ORB PORT") + "/" + name;

    return url;
}

dshop.commerce.shopping.everything.NamingServerImpl server =
(dshop.commerce.shopping.everything.NamingServerImpl) obj;
switch (opnum) {
case 0: // getURL(String)
{
    java.lang.String $param_String_1;
    try {
        java.io.ObjectInput in = call.getInputStream();
        $param_String_1 = (java.lang.String) in.readObject();
    } catch (java.io.IOException e) {
        throw new java.rmi.UnmarshalException("error unmarshalling arguments", e);
    } catch (java.lang.ClassNotFoundException e) {
    
    } catch (java.io.IOException e) {
    
    }
}
throw new java.rmi.UnmarshalException("error unmarshalling arguments", e);  
} finally {
    call.releaseInputStream();
}

java.lang.String $result = server.getURL($param_String_1);
try {
    java.io.ObjectOutput out = call.getResultStream(true);
    out.writeObject($result);
} catch (java.io.IOException e) {
    throw new java.rmi.MarshalException("error marshalling return", e);
}
break;
}

Of the source generated by the RMI compiler, we only included the dispatch() method, which is the worker method responsible for forwarding method calls to the server object. Furthermore, we wrote down the important code in italics. A first reading of this code clearly shows that the server-proxy performs the following operations:

- It reads the hash code for the method that the client wishes to invoke
- It compares this hash code with the ones it knows – exposed by the RMI framework, a code for every method
- It creates an instance of the server object to forward the method calls to it
- If the proxy knows the method that the client wishes to invoke, it calls that method on the server object it just created. The parameters of the method are read from an InputStream, which clearly reminds us that RMI is using a TCP/IP socket stream, something that is not very unusual for a Java network technology
- After the method reads all the parameters that have been marshaled to it, it invokes this method – after doing the proper casting – on the server object, and collects the result
- It then forwards the result back to the client-proxy, writing it back to a ResultStream.
Socket communication is very easy in the Java language. RMI therefore transforms method calls and parameter operations into bytes moving to and from nodes on the network. Although the example we discussed here uses primitive Java types as parameters, it is possible to use custom objects, for which a delegated model is used to request the representing bytes. RMI does that through a specialized class-loader, which is called the \textit{RMIClassLoader} at the server node. In all, RMI makes sure the methods are properly invoked and their results sent back to the client, otherwise it generates an exception, which the client should catch and handle accordingly.

\subsection*{2.2 DCOM}

The Distributed Component Object Model evolved as the Microsoft way of marrying two very demanding technologies at the time. While the component object model was finding its way into almost all application developed using the \textit{Windows} authoring development environments, distribution was being attempted by these application authors through common but non-standard network technologies and protocols. Very soon the multiplicity of distribution algorithms and methods started to create compatibility problems not only to different third-party vendors among each other, but also to the same vendor products. In addition to this compatibility problem, the continued support of these distributed protocols meant that enterprise resources, including development and maintenance, had to be split between the application logic and the networking technology.

The market demand for a new Microsoft-supported distribution technology added to the competition from already popular distributed technologies that were driving enterprise applications, some of which were running on Microsoft operating system platforms. The solution was DCOM, which took COM as the basis of its operability. DCOM was simply the smart way out for Microsoft. COM was already well established in most applications, and was the primary object technology in the market. Microsoft saw COM as the single-node version of a bigger thing. When an application \textit{invokes} a method on a COM object, it does not know how and where from this object is loaded, and hence it could very well be localized at some other machine for all that application
knew. And with Microsoft playing the home-side game – so to speak, the entire operating system was at its service, and they knew what to do with the advantage they had. Microsoft built DCOM straight into the registry of it Windows operating system. This is sometimes overlooked when talking about DCOM in technical literature. The DCOM architecture is always described as service-driven. By this it is meant that requests from remote machines are forwarded to a local service at the remote machine, which communicates with the remote service on the remote machine.

The best way to understand how DCOM works is to mirror it to the other distributed technologies that we study in this thesis. The general distributed model requires a centralized “lookup service” that tells the clients where to find the object that can provide the requested service. It is worth noting that unlike all other distributed object technologies, DCOM has no such lookup service. Even WebServices, the technology considered as being the most loose, has a service lookup directory that, although might be located outside the network itself, still provides dynamic object lookup for the running clients. The lack of such a service in the DCOM architecture is due to the requirement set by Microsoft that DCOM should be 100% compatible with COM, that it should default to COM on a single-node cluster, and that DCOM should use Windows architectures to lock this technology into the Microsoft closed-source community, and to make it impossible for third-party developers to leverage the powers of DCOM away from Microsoft-driven technology. Indeed programming a DCOM application is probably the easiest procedure one can think of as compared to all the existing distributed technologies. The simple development road map for DCOM applications is the following:

1. Build a COM application locally
2. Test the application on the single-node server
3. Identify the COM objects that need be deported to a different server
4. Go to the registry and change the entries for these COM objects to point them to the remote server

Notice that this is a general procedure. The steps are probably implemented through automation tools, and the developer might not notice where and how things are done. Another thing to notice is that coding and developer intervention stops as of step 2.
This alone is a major cutback in coding and development. No other distributed technology can offer such a rapid distributed application development model. With the Microsoft authoring and development tools that integrate the application logic with COM from the early stages of the application’s development, the DCOM model became a major attraction for almost all the application development teams.

DCOM offers this convenience at a very big price. Client-registry changes are required every time the distributed cluster is modified. This might be a rarely performed operation, but when it is time to do this, all the client’s registry entries need be modified, since now the object has moved. It is possible – and in such a context desirable – to create application-layer registry management logic that would query some “fixed” remote server that parallels the naming service in the other distributed technologies. The server-management team would publish the locations of the objects at this server, and the clients would have to change their registry every time such a location changes. The client user would not notice this change, but it is still being performed on every client. While this procedure sounds complete from a logical point of view – it is a simple lookup and modify procedure – we are not aware of a standard Microsoft tool that does this, probably in an effort of keeping DCOM as simple and technically “close” to COM as possible, thereby benefiting from any changes that might be added to COM.

Finally, DCOM is not recommended for Java applications. DCOM makes Java lose the portability that made Java so popular and adopted by enterprise application developers. Moreover, DCOM locks the Java applications to the Microsoft framework, and drives the object-oriented programming model into the COM framework. Since COM is the object-oriented model competitor, DCOM had to pose as the distributed object software competitor. But COM lost the battle long ago. COM had to be made so simple that applications written in the Visual Basic programming language – a language that has no object-oriented methodologies – could author and use them. As a matter of fact, COM was used by Microsoft to enable non-Java applications to access Java objects locally. Therefore DCOM allows non-Java applications to be written that would use Java objects at remote locations, and the opposite is also possible as well. This marriage between COM and Java is bad, and the proof is no-where other than at the core of the Microsoft technologies. While the Java language’s library classes where growing in all
the right directions, and the enterprise-driven improvement in the language’s extendibility and programming model was putting new technologies at the heart of the language, Microsoft’s JVM was limited in its ability to compete. Microsoft saw that this competition was not worth the effort, and promoting Java-driven components was nothing in the interest of Microsoft. COM’s integration with Java was meant to move Java developers into the COM framework, and not otherwise, and Microsoft had already developed a transaction server that was driving the enterprise.

In conclusion, and like all other distributed object technologies, DCOM has its benefits and drawbacks. But it so happens that DCOM builds its benefits on the success of the Microsoft operating systems and object technologies, but we do not feel that components using this technology allow application development to grow properly, since it lacks the proper object-oriented attributes. DCOM is good when living inside the Microsoft atmosphere, but inside that atmosphere the only competitor of DCOM is another technology that threatens to replace it. Microsoft’s .Net architecture cut the last rope that held DCOM together all this time, and it is now almost certain that Microsoft is going to halt COM development very soon, if it had not done so already, and thus DCOM will very soon be replaced by the .Net architecture.

2.3 CORBA

CORBA was the answer to the old question that would get asked every time a distributed application was developed. That question was whether an intra-language solution would be desirable that could allow components written in one programming language to communicate with other components written in a different one. This problem had come to occupy the minds of people working with distributed solutions from many angles. The first angle was that many components had been written without distribution in mind. These could have been single-node components that served well at that level, and when distributed applications were to be written, such components would either have to be ported to the new language or interfaced through it. But this was not the only problem, or otherwise it would have been possible to come up with intra-language communicating components on the same node. But what added to this was the
requirement that these components be located at specialized nodes in the cluster. This meant that interfacing such component would have to achieved at the cluster level rather than the language level.

The CORBA answer was to create a new language, the Interface Definition Language – or IDL. IDL would instruct such components on how to expose their methods – or functions – and this would allow objects to communicate by marshalling their method calls to the broker architecture that CORBA held, and then that would lookup the objects and their server-node’s location, and the method parameters would be marshaled. In doing this, IDL would be to CORBA as any language is to its operating system – or virtual machine in the case of Java. IDL defined many language translations and transformations, and it grew to encompass many other features such as exceptions and user-defined types. We will not discuss the syntax of IDL in this thesis, since it is available in the literature and across many technical references. We will, however, explain some of the aspects that make IDL an excellent choice for such object broking architectures. Indeed IDL became the de-facto standard for object communication specifications, and many ORB technologies – like RMI, while not requiring IDL-specified language transformations, support IDL and allow applications and components that export themselves to the naming server for distribution to be translated into their native IDL form. IDL offered a layer on top of the programming layer, and that meant that a new stage had to be inserted into the distributed programming model – IDL to native language translation. Such a layer or stage would be redundant in only one situation; when the two communicating components were written in the same programming language and that language supported object distribution. A very famous example is Java, which has RMI at it the core of its programming model. But even in that situation IDL translation would be desirable, since it would enable application developers to experiment with the components they build, replacing them with others written in a different language to leverage the power of the underlying operating system – as is the case with C++ components. The extra layer would have to statically bind to the application, since run-time translation is bad for performance, to say the least. To avoid such a static linking, dynamic object binding was introduced into CORBA that would allow objects to expose their properties and methods when they are queried or otherwise
when introduced into the cluster. This meant that no IDL was required, at least at the development stage.

2.4 WebServices

With all the available ORB technologies and network protocols, the appearance of another such technology was not desirable. This statement was correct up until the Internet grew to an un-controllable forest of intranets, most of which would not be trusted. In addition to this, that forest soon became the only place where business was conducted. Many data-driven resources and applications where now hosted at some dedicated server thousands of miles away from the headquarters. And to make things even more complicated, businesses started emerging that depended on nothing but the Internet to deliver their services. Many communication channels were deprecated – so to speak – and forwarded to some website. With the growth of the Internet came the question of whether such a huge forest of information can communicate and how it would so. To really appreciate the importance of WebServices as "the" solution to this problem, it is important to have an understanding of the demands set for such a technology. At that time there were only two conceptually distinct ways for applications to communicate over a network; through insecure intranet technologies such as RMI and CORBA, and through secure but unreliable Internet technologies such as CGI and server-side scripts. It was realized that a technology that can put the good features of these two into a productive and cost-effective solution would be highly desired. Object technologies that had been running enterprise applications lacked many security features, and could not be used over a wide insecure network like the Internet because of the nature of their communication methods, which relied on certain network protocols that were not supported at this large scale. In addition, the growth of insecurity brought with it a new breed of security precautions in the form of firewalls and bogus IP addresses. With the Internet it was possible to introduce servers that would have not visibility and still be operational and use services on other visible networks. Such a situation – for example – would definitely fail any object technology. But while these factors were weakening
object technologies at the Internet level, the performance and reliability factors were
doing worse to the Internet technologies in the smaller arena of the enterprise.

WebServices were all that was needed to get things done – securely! Just like a
person would browse a website using the commonly used Internet protocols like HTTP
and FTP, an application would browse another application using WebServices, which
itself was built on top of these protocols. The same delegation model that is used with
object technologies to enable service-lookups was used with WebServices. UDDI – or the
Universal Description, Discovery and Integration – can be used to forward lookup
requests to naming servers that would then retrieve the location and call-parameters for
the application at run-time. This put WebServices in the competitor’s seat, and to add
more challenges to this, it supported secure connections, protocol-level authentication,
and with the parallel growth of communication speeds and reliability, WebServices were
fast and reliable enough to make it to the enterprise. It was possible to write WebServices
in any language that one could think of, and web servers were being prepared to host
these services, providing server-side coding and run-time flexibility. In all, WebServices
were the preferred way to go about building distributed object software that extends its
functionality to the boundaries of the Internet. It was there that all other competition
simply ended.

But how do WebServices work anyway? Simply stated, an XML-based request in
a standard format would be packaged inside an Internet usable protocol – such as HTTP
or FTP or SMTP – and the request would be made to the server where the service runs.
The location and other call-parameters required to make this request should be either
known prior to the application launch or detected at run-time through UDDI.

WebServices would get published to servers through the use of WSDL – the Web Service
Definition Language. This is also an XML-node that describes what the WebService does
and how it should be invoked. Whatever the methods, the request will be sent over the
Internet just like any other request to view a web page or read an email. Web servers
would treat this request in the same way as they would any similar request, and the
content package would be forwarded to the WebService, which would read the request
and analyze it – probably decrypting it first – and the object reference, method name and
parameter types would be looked up and resolved, the object could then be either
specifically created for this request, reactivated for the current user session, or simply forwarded to an already active instance in the case of application-level services. The three types that we just mentioned are the three types of WebServices that can exist at any server – namely request, session and application wide webservices. The difference is sometimes very critical to the success of the software, and care should be taken when choosing the appropriate type.

After the object or instance is provided with the parameters for its method to execute, the result is then forwarded to the client – provided the client is still there after it requested the service. The same marshalling procedure is used, and serialization is used to send instances as parameters and return types. This marshalling procedure is similar to that used in CORBA, and it offers an additional layer on top of the existing network layers since services written in one language could be interfaced by clients written in another language. The exact format of the XML-driven requests and responses can be found in the literature and the WebServices’ specifications. What is important, however, is that humans or applications themselves rarely process this format. Special API’s are used at both sides of the service channel, creating the XML node at the client, forwarding it to the server, and analyzing it there to retrieve the method and call-parameters.

Therefore WebServices are programmatically simple to use, and almost intuitive. To show how simple it is to build a webservice request, we will show an XML-node that is needed to invoke a method at a remote server. The code is adapted from the Internet resource found at http://www.cs.une.edu/Courses/comp190/docs/lessons/ws/ws_soap:

```
POST /axis/MessageObject.fws HTTP/1.0
Host: localhost

Content-Type: text/xml; charset=utf-8
SOAPAction: ""
Content-Length: 448

<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding"/>
```
<soapenv:Body>
<ns1:getMessage xmlns:ns1="http://soapinterop.org"/>
</soapenv:Body>
</soapenv:Envelope>

The response that this request would get is again similar:

HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Date: Tue, 17 Sep 2002 15:35:36 GMT
Server: Apache Tomcat/4.0.4 (HTTP/1.1 Connector)
Connection: close
Set-Cookie: JSESSIONID=115C2B8879FCC1E2693BE3F9517D7910;Path=/axis

<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

<soapenv:Body>
<ns1:getMessageResponse
 soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
 xmlns:ns1="http://soapinterop.org"/>
<getMessageReturn xsi:type="xsd:string">Hello, World, from jbs MessageObject</getMessageReturn>
</ns1:getMessageResponse>
</soapenv:Body>
</soapenv:Envelope>

Notice how the server creates a session id when this service is requested. If the service is of a session type, this id will play an important role in keeping track of the client and what instances it used. Notice also how the SOAP request embedded into the HTTP message.
3. Proposed Evaluation Approach

To properly understand the behavior of each of the ORB technologies discussed in the previous chapters, we need to perform predefined operations within the framework of each technology and then analyze the results obtained and try to answer some questions common to the technologies under study. Every one of the technologies being studied is an object request broker, which means that it is supposed to enable object communication in a distributed cluster of networks. Such objects are usually part of a distributed application that relies on one of these technologies to expose its services. Having such an application is the only way the various properties of these technologies can be evaluated and understood. Therefore it was important for us to build such an application, making sure that the same abstract object communication is performed regardless of the implementation and the ORB technology used to achieve it.

In this chapter we will discuss the distributed application that we built to study these technologies. The fact that these technologies facilitate object communication in a clustered environment, we found it very useful to build an enterprise application that relies solely on each technology to do the same cluster-wide operation. There was also the need to test the underlying technologies for various properties, and therefore the application had to implement the logic required to enable these properties or services for each of the technologies under study.

In order to understand how our application works, its components are detailed and explained, revealing the various classes used to make the application work. It can be seen from this structure that the application’s abstract logic does not rely on any particular object technology, but rather abides by the general approach for building distributed software. The technologies are expected to handle the application’s classes and enable invoking methods on them.

Such criteria are common to the ORB technologies being studied, and criterion evaluated by the application is compared across the technologies. Some criteria of distributed technologies cannot be evaluated empirically, and are thus discussed from a qualitative point of view. These criteria and the empirical ones are detailed in this chapter and their definitions drawn from our experience with these ORB technologies.
3.1 Overview

Evaluation of distributed object technologies is not a straightforward procedure. Indeed, finding the right configurations within which the evaluation is performed is very critical, and the choice of application architecture to reveal the powers of these technologies might make all the difference between a successful measurement and a wrong evaluation. In this section we will discuss the general framework that we follow to successfully generate our evaluation criteria's index values. We will describe the framework both from a technical and a structural perspective, and we will try to suggest directions that can be followed to include additional criteria and perform the necessary measurements within this framework.

Before we discuss the enterprise application that we constructed, we will detail the inner structure of the framework within which this application runs. As is obvious from every application development experience, building applications from ground up is a very tedious process, but it involves many repetitive steps that can be filtered out and included in a general framework. This methodology is not particular to distributed software, but is rather a general software engineering approach. The libraries that emerge from iterative development processes help reduce the time and cost in enterprise-level applications. In distributed object software development, the libraries grow even wider with the inclusion of communication objects and protocol-processing objects. We will discuss in later sections the problems and complications associated with such objects, but it is obviously a very intelligent decision to add such objects to a framework that makes them usable elsewhere across the development process. But there is a difference between a repository of object components and a framework that utilizes their dependencies to reduce development and deployment times. We followed this approach and decided to come up with a framework that not only encapsulates the common libraries used across the same versions of the application, but also drives future enterprise applications that themselves add to the framework, and thus it continues to grow across multiple enterprise attempts, making the framework a powerful tool for distributed application development using the technologies under study.
But the framework had to be built from scratch in this thesis. The application used was a challenge in itself, since this is one of the rarest attempts to do the same kind of work using different distributed object technologies. While a similar framework would have evolved rapidly had the application used a single technology, our framework had to join together structurally different object technologies that required configurations prior to the application launching. The common denominator had to be taken from these different versions, and objects that had great importance in the working of certain versions had to be ruled out because of their lack of reuse with different technologies. We found that a separation had to be made between utility classes and communication classes, and that is why we divided our framework into a common utility package and an integration package that consists of networkability and analysis sub-packages.

In addition to these packages, the deployment of our application had to be performed across multiple networked machines, and once deployed the application had to benefit from a centralized configuration server that provided it with dynamic property values required by the application in order to function properly. Such properties included database configuration parameters, network locations and operation-load scales. Thereby we devised a general daemon that launches our different versions of the application in varying configurations. The daemon is started at every cluster machine, and at every non-central node that this daemon is running; the configuration is delegated to the central server for localized property management. We also initiated configuration layering that would replace network configuration files with database-driven ones. This gives us the choice of storing the properties in both tables and flat files. We found that TCP/IP is much faster in a Windows network than other Microsoft-devised protocols, but network files are much easier to edit and change since the other alternative would include database-interfacing applications. Whichever the configuration choice, the application would still work as expected.

In what follows we will describe in detail the preliminary structure of our application's classes. This description is not conclusive since other classes in the framework are present and do the work that is required by all similar applications. The discussion of the framework classes is very lengthy, and we recommend reading the code, which we tried to make as simple as possible.
3.2 Enterprise Application

In order to properly experiment with the four ORB technologies that are the target of this thesis, we develop a distributed application using the three technologies. In this subsection, we discuss our enterprise application and try to point out what we call the communication points, the points in the code where the objects are expected to use the ORB to delegate their communication.

Figure 1: Application Components

3.2.1 The Idea

The choice of what the application should do was almost arbitrary. We needed an application that tries to solve a problem that is in its essence distributed. In other words, we looked for a problem that usually requires more than one execution location to solve, and in this search we came upon the idea of our enterprise application.

We called our application “Distributed Shop”. The application emulates a shop with a stock distributed over various locations, and with a management that is also
distributed. The stock might reside on different servers for various reasons. It might be the case that storage space is limited, or for transportation reasons the shop owners decided to place certain items in locations close to the manufacturing company. Whatever the reason, the various shops that we looked had a distributed stock, and it made sense to pursue in the logic of our application.

3.2.2 The History

The actual application that we used to collect data to study the various criteria in our evaluation was not the first attempt that we made to come up with a distributed object application. We started with a rather bold attempt, a trial application that can be configured at run-time to use one of the four ORB technologies. We were hoping to create a foundation on which we can build any distributed application and tune the application to the ORB technology under study. In our attempt we tried to build common classes that make-up the basic framework through which any application can be developed. We called this framework the Object Communicator System, or simple OCS. We faced many problems when moving forward with OCS, and we tried to solve every problem that we faced, but eventually the framework came to a dead-end, and we had to go back to the drawing board, starting from a simple two-class single-node application that we cloned to create the four versions for the four ORB technologies. We tried to make the changes in the application minor when the ORB code was inserted.

We learned from the first attempt at a unified framework that we cannot unify ORB technologies not even at the code level. But we did gain invaluable experience from this attempt, as we were able to plug-in the ORB-specific code in a relatively small time and with minimal effort when we developed the second application.

3.2.3 The Technology

DShop uses every one of the four ORB technologies to perform a certain computation. The five versions of the application (single-node version and four ORB-compliant versions) perform the same basic operations and computations. The differences between the versions reside in the core communication classes, in addition to some
initialization code. These differences are isolated and can be removed at any time, collapsing the software to a single-node application.

3.2.4 The Design

In order to arrive at a distributed application that could realize the idea presented earlier on, we had to come up with a design that could fit all ORB technologies and serve as a starting point for the development procedure.

The design of DShop started with the single-node application. We had to define the goals that we wanted to achieve, and we came up with the following list:

1. **The Business Logic:** Since we had little knowledge of the internal workings of a grand-size shop, and since we had no intention of growing the application in its internal logic to serve as a solution, but rather as a means to evaluate ORB technologies, we decided to scale down in logic. The business process that we decided to implement was a stripped-down version that fits the cart-counter-stock shopping process. The application does not support many features that we feel are necessary for a real shop, but the door is open for future work, and what we did serves as a good starting point.

2. **The User Experience:** Another goal that we deemed important was the way the application interacts with its users or customers. We wanted the application to have all the functionality necessary to process customers with their carts filled with products. Since the application would be tested in an experimental computer lab, we had to automate the process of generating the resulting data. The application would be fed input files generated by another application, and these would act as customer lists with their corresponding carts. With distribution in mind, and having very limited human resources, we aimed for the automated solution.

3. **Database Support:** We had to decide before starting the design on the source of information for the application. We had in mind four sources of information to decide upon:

   a. **Customer Carts:** each customer should have a list of products that he/she wants to check out of the shop (or buy). This list – or the carts – had to be
available at the client machines, and the information should be read by a
client manager that initiates the shopping experience for every customer in
the list. Having decided on the location of the carts, we found it
impossible to install a database server on every client machine, and
therefore we decided to store all the carts of all the customers on any client
machine in one data file. The customer name would appear at the top of
the list, with the list of products and their corresponding quantities
appearing on separate lines beneath the customer name. A delimiting line
would separate the carts.

b. List of Products: Before we could decide how to represent the products,
we had to decide on what machines to store the names of the products,
their prices and their identification codes. Surely the server machines had
to have copies of the products to work with. But would the client machines
need the products. The answer was no, because the client machines would
read the list of products from the carts, and since this was an automated
procedure, we decided to keep a copy of the products only on the server
machines. We also decided to keep the products’ list as read-only.
Therefore the names of the products would not change, their pricing would
remain the same, and their codes would be kept intact. With this decision,
it was now possible to replicate the product lists on all the server
machines. This would reduce the network load, since servers need not
communicate product-specific information. Finally, we decided to store
the products in a table in a database server installed on every server
machine.

c. List of Customers: A reasoning similar to that used with the products had
to be followed with the list of customers. The customers would not be
allowed to change, and their information would be replicated over all
server machines. Therefore we represented the customers in a table in the
database.

d. The Stock: The stock is a repository of products. It contains all the product
items that the shop can provide its customers. The stock should initially
contain enough items of ever product to serve the customers, and as the stock reduces in size, an external source tries to balance the stock. In our case we do not have an external source, so the stock only reduces in size, and the shop may refuse certain customer requests if not available in the stock. It is very easy to simulate an external source to keep the stock balanced, but this would not have affected the evaluation in any way, as the same extra process would have been used over all five implementations. Unlike any of the other three sources of information presented above, the stock is not a read-only source. The application should be able to write to the stock, and change or delete certain products from it. And we wanted to distribute the stock over the servers on the network. Since the stock can be easily partitioned, we decided to place parts of the stock on every server, and the stock would be represented as a table in the database, with the product identifier and the quantity on stock as fields.

It is worth noting that the database server that we used, namely MySQL, makes it possible to access an instance of the database running on a server machine from another machine on the network. Therefore it was possible to have only one database server and have all servers access this server over the network. But we chose to have copies on all servers and limit the application instances to the local database server instances for a couple of reasons. First, we did not want the database server to become the bottleneck in the system. If only one server were to answer all queries, it would have limited the performance dramatically. Second, as in all Java applications, we are using JDBC for database connectivity. We wanted our application to work regardless of the database server used. Therefore we should be able to replace MySQL with any other database server and not change anything in the application. If we later decided to use a database server that had no network support, the application would not work. Third, we wanted to limit the network traffic to the technologies being tested. Allowing the database server to use network would have added a factor to our evaluation that we have little control over.
3.2.5 The Class-Diagram

The customer carts are loaded, the carts forwarded to the shop, the shop loops over the carts adding up the price of every product weighted by the quantity requested, and finally returns this total price of the cart to the customer. The shop should reduce the stock by the quantities requested every time a cart is processed. Most of the classes involved in the class-diagram are an implementation of an interface. The interfaces – client and server – are placed in a separate location and separated from the implementations. This is important in a distributed application. Although this is still a single-node version, it makes it easier for us to extend it to a distributed application with the server, client and interface files in separate locations.

The following interfaces define the single-node version of DShop:

3.2.5.1 ClientManager

This is the only manager on the client-side. The ClientManager is responsible for reading the customer carts and forwarding them to the shop for processing. The following are the methods or operations that the client manager is responsible for:

1. `nextClient()`: the implementation of this method should try to prepare a customer cart to be forwarded to the shop.
   
i. **Prerequisites**: the customer carts’ data source must be open and ready for loading the carts. The method will not attempt to re-open this source if it fails to read from it.
   
   ii. **Post requisites**: when this method returns, a customer cart would have been forwarded to the shop. The method does not guarantee that the cart has been properly handled.
   
   iii. **Parameters**: this method takes no parameters.
   
   iv. **Return Value**: the method’s return value is of a “boolean” type. It is *true* if it succeeds in reading the next customer’s cart and forwarding it to the shop. Otherwise it will return false, which could either mean that no more customers are available for the client manager, or that the particular customer cart failed to load.

2. `start()`: this method is responsible for starting the loading of customer carts.
i. **Prerequisites:** the process of loading the carts is supposed to have been stopped or never started in the first place. The method will not attempt to stop this process. It will continue calling `nextClient()` until either the latter returns `false` or the process is stopped.

ii. **Post requisites:** after the method returns the client manager would have either processed all the customer carts available or stopped abruptly at one of the carts. The process of loading the carts is never interrupted and resumed with the same cart. This means that once a cart is read and forwarded to the shop, it will never be processed again.

iii. **Parameters:** the method takes no parameters.

iv. **Return Value:** the method does not return any value to the caller.

3. **stop():** this method tries to stop the process of loading the customer carts. A call to this method guarantees that the next cart will not be processed.

i. **Prerequisites:** this method does not presume that the cart-loading process is running. In this sense, the method can be called any number of times, regardless of the state of the client manager.

ii. **Post requisites:** a call to this method does not guarantee that the cart-loading process will stop. In fact, this method can best be described as a request method, in that the caller requests the stopping of the process rather than forcing it to stop. It is up to the implementation of the `ClientManager` interface to decide on whether the call to this method has any effect on the process.

iii. **Parameters:** this method takes no parameters.

iv. **Return Value:** the method does not return any value to the caller.

3.2.5.2 Cart

This is the actual wrapper for a customer's cart. An instance of the implementation is created by the client manager, and in particular through a call to the `nextClient()` method. The cart represents an entry in the customer carts data source.
1. **add()**: this method is called whenever a product item is to be added to the cart. The ClientManager usually calls the method during the cart-loading process, but it is not the only way to load the cart with the items.

   i. **Prerequisites**: the cart should have enough space to hold the quantity of the product item being added. The method will not attempt to check for the availability of this space.

   ii. **Post requisites**: after this method returns, the cart's content of the given product item would have been increased by the specified quantity.

   iii. **Parameters**: the method takes two parameters.

      a. **item**: the product item to be added to the cart. This is of a *string* type

      b. **quantity**: the actual quantity of the item to add to the cart. This is of an *integer* type

   iv. **Return Value**: the method does not return any value to the caller

2. **checkOut()**: in this method the cart is checked-out with the shop. The checkout process involves identifying the shop that the client or customer wishes to buy the quantities of product items in his/her cart. The shop should then compute the total price of the cart, and in its own way inform the customer of the price. This method encapsulates this process, and returns the price of the entire cart.

   i. **Prerequisites**: this method is a forwarding method. It does not perform computation of any kind, but rather forwards the cart content to the shop that – through its *Shop* interface – performs all required computation.

   ii. **Post requisites**: the cart does not undergo any changes when this method returns. Although the expected behavior of this method is expected to clear the cart, it is not

3. **remove()**: this is the opposite of **add()**. It is supposed to remove a specified quantity of a product item from the cart.

   i. **Prerequisites**: when invoked, the method tries to perform the removal regardless of the presence of this product in the cart. Therefore there are no prerequisites for the method call.
ii. **Post requisites:** when the method returns, the cart would have been reduced in size either by the quantity requested for removal or by the already existing quantity in the cart

iii. **Parameters:** the method takes two parameters.
   a. **item:** the product item to be removed from the cart. This is of a **string** type
   b. **quantity:** the actual quantity of the item to remove from the cart. This is of an **integer** type

iv. **Return Value:** the method does not return any value to the caller

### 3.2.5.3 ServerManager

This is the first of the server-based classes. This class has been assigned the task of managing server-to-server communication. In fact, this is the only class in the system that is aware of the presence of other servers on the network. The model we are trying to use is simple. All other components of the application will forward their processing tasks to this class, which in turn will try to distribute the workload over the available servers on the network. This way, the application logic is separated from the distribution logic, making it possible to extend the application to involve many other classes with new operations and business logic, but still maintain a standard framework through which distribution of the application is managed.

Our distributed model has one master server and a multitude of slave servers. We borrowed the slave-master concept from parallel programming, in which a master node controls the flow of data to the other slave nodes. In our model, however, the nodes are server nodes, and we have taken this concept one step further in separating the client activities from the server activities. All servers on the network will try to work together to serve all the clients.

In order to perform its distribution tasks, the server manager requires that the participating classes in the system expose two methods for forwarding purposes. The server manager should be able to query every server on the network – including itself – to find out if it can perform the operation at hand, and then request this operation from that server. Every interface in the system should expose two methods for every operation. The
first method is a query method — with its name starting with the prefix ‘can’ — and the
method that actually performs the operation.

No matter how many clients the shop is processing, and no matter how many
threads are running on the server node, there should only be one server manager instance.
For this reason, all other classes that require the server manager are urged to create a
static instance reference and use it in conjunction with distributed operations. It is for this
reason that all methods in this interface are expected to initialize the server manager if it
has not been initialized. This is required because the constructor of a static reference is
invoked only once during the lifecycle of the application.

The interface for the server manager allows classes to call the following methods:

1. **isMaster()**: this method detects whether the server hosting the instance of this
   interface’s implementation is “the” master server or a slave server. It is very
   uncommon in our framework that any other class should query the server manager for
   this information.

   The method is provided for querying the underlying server, and the only thing that
   it looks for is a file called “servers.txt”. This is a text file that includes the network
   addresses of the servers involved in the distribution. Only the master server has access to
   this server, and this method looks for this text file to know if the instance is a master
   server.

   i. **Prerequisites**: the method does not assume anything about the state of the
      server manager. It will however try to initialize the manager if not initialized.

   ii. **Post requisites**: since this is a querying method, the state of the manager is not
       altered after the method returns.

   iii. **Parameters**: the method does not take any parameters.

   iv. **Return Value**: the method’s return value is of a **boolean** type and is **true** if the
       server node is a master server and is **false** otherwise.

2. **canPullOut()**: this is the query method of the pullout operation. The stock manager on
every server node should have a similar method that this server manager invokes.

   i. **Prerequisites**: the method requires the presence of a stock manager instance
      prior to the invocation. This condition is guaranteed through the initialization
      of the server manager, which this method attempts to perform.
ii. *Post requisites:* being a query method, this method does not change the state of the server manager.

iii. *Parameters:* the query involves two parameters:
   a. *productid:* this is the product identifier that the method is querying for pullout. The parameter is of a *String* type.
   b. *qty:* the quantity of the product to pullout. The parameter is of an *integer* type.

3. *requestPullOut():* this is the forwarding method for the pullout operation. It forwards the pullout request to the least busy server on the network through a call to the *pullOut()* method of the stock manager on that server.
   i. *Prerequisites:* if the server instance is a slave server, the method assumes that the local stock manager has been initialized. However if the server is the master server, it requires an initialized and ordered list of the participating servers.
   ii. *Post requisites:* after the method returns it would have either forwarded the pullout request to the proper stock manager, or otherwise failed to complete the request. No matter what the result, the method does not alter the state of either the server manager or the stock manager involved.
   iii. *Parameters:* the method forwards the parameters of the pullout request to the stock manager:
       a. *productid:* the identifier of the product item to pullout. This is of a *String* type.
       b. *qty:* the quantity of the product to pullout. This is of an *integer* type.
   iv. *Return Value:* the method does not return anything to the caller.

4. *canCheckOut():* this is the query method of the checkout operation. The method forwards the query to the customer manager on the local server.
   i. *Prerequisites:* the method requires that the local customer manager be initialized before it forwards the query. Since the customer manager is initialized with the server manager, it attempts to initialize the latter.
   ii. *Post requisites:* the state of the server manager is not altered after this method returns.
iii. **Parameters:** the method uses the parameters of the checkout operation:

   a. **clientid:** the identifier of the client to checkout. This is of an `integer` type.

   b. **items:** the items or list of products to query for checkout. This is a comma-separated list of the product identifiers, and therefore it is of a `String` type.

   c. **qty:** also a comma-separated list of the corresponding quantities being queried for checkout. This is the actual content of the client’s cart. It is of a `String` type.

iv. **Return Value:** the method returns the answer of the customer manager. Its return type is `boolean` and it is `true` if the customer manager can perform the checkout operation for the given client. Otherwise it returns `false`.

5. **requestCheckOut():** this is the forwarding method of the checkout operation. On the master server node, the method – in a behavior similar to that of `requestPullOut()` – will attempt to locate the server with least load and whose customer manager can perform the checkout operation, and then forward the customer’s cart to it. On a slave server node, the method will forward the request to the local customer manager directly.

i. **Prerequisites:** the same prerequisites of the `requestPullOut()` method apply to this method. The server manager should have available an ordered list of the participating servers. On slave server nodes the method also requires the instance of the local customer manager. Both of these requirements are met when the server manager is initialized.

ii. **Post requisites:** the method neither alters the states of the server manager, the local customer manager, nor the remote customer manager instances.

iii. **Parameters:** the method uses the checkout operation’s parameters when forwarding the request:

   a. **clientid:** the unique identifier of the customer being checked out. This is of an `integer` type.

   b. **items:** a comma-separated list of the product items’ identifiers that the customer wishes to buy. This is of a `String` type.
c. *qtxs*: also a comma-separated list of the quantities corresponding to the requested products. The type of this parameter is *String*.

iv. **Return Value**: the method does not return any value to the caller.

6. *serverLoad()*: this method is provided for the purpose of achieving true dynamic load-balancing. The method is invoked on the participating servers by the master server manager instance.

i. **Prerequisites**: the method will attempt to compute the server load. This operation requires instances of both the local customer manager and the local stock manager. Again this requirement is met when the server manager is initialized.

ii. **Post requisites**: the method will query the local managers, and therefore no modification is made.

iii. **Parameters**: this method does not take any parameters

iv. **Return Value**: the method computes the server load in the form of an *integer* value.

3.2.5.4 *CustomerManager*

The customer manager is the server component responsible for checking out customer carts, in addition to other customer-oriented tasks. The manager uses the customer data-source to answer queries and perform checkout operations.

The customer manager collaborates with the local stock manager. It also collaborates with the server manager to perform the checkout operation within the load-balancing framework.

1. *manageCustomer()*: this method is the starting point for the managing process. The actual work performed depends on the implementation, but the method is expected to lookup the id of the customer. This unique identifier will be used by the shop instance to identify the customer.

i. **Prerequisites**: the method will perform database queries and therefore needs a usable database connection. This is provided by the static methods of the *ConnectionManager*.
ii. **Post requisites:** after the method returns the customer’s unique identifier would have been looked up.

iii. **Parameters:** the method only needs the name of the customer:
   
a. **customer:** the name of the customer, which is then used to lookup the id of the customer. This is of a *String* type

iv. **Return Value:** the method returns the identifier for the customer in the shop. If the customer is new, it creates a new identifier and returns it. The identifier is of an *integer* type.

2. **checkOut():** the method is the worker method in the system. Many other collaborating classes forward the cart to this method. The *checkOut()* method adds-up the cart’s content and computes the total cost of the customer’s cart. After computing the total price of the customer’s cart, it contacts the stock manager requesting the pullout of the product items from the stock.
   
i. **Pre-requisites:** the method processes the cart’s content through the product items’ list and the quantities list. Both of these lists have to have the same length. The *Shop* is expected to prepare these lists in a consistent way. In addition, the method requires a working database connection, which it uses to lookup product prices. The method also requires a valid stock manager instance. The latter is prepared when the customer manager is initialized.

ii. **Post requisites:** the method performs the cart checkout operation and makes sure that the stock manager is notified of the items requested. It does not guarantee the successful removal of these items from the stock, nor does it ensure the availability of the items in the first place.

iii. **Parameters:** the method uses the cart checkout parameters as follows:
   
a. **clientid:** this is the unique identifier of the customer being checked out. It is of an *integer* type.

b. **items:** a comma-separated list of the product items requested by the customer. The list is of a *String* type.

c. **qtv:** an equal-length comma-separated list of the corresponding quantities requested. This is also of a *String* type.
iv. **Return Value:** the method returns the computed total price of the cart. If the cart is invalid or if the method fails during the computation, the returned value is zero. The type of the returned value is **double**.

3. **canCheckOut():** this is the query method that the customer manager provides to facilitate the server load-distribution. The method is the worker method invoked by the **canCheckOut()** method of the **ServerManager** interface. The method should find out if the server node can handle the new customer and follow-up through the entire checkout operation.

   i. **Prerequisites:** the method requires a valid customer cart. The state of the customer is irrelevant, and the method can be invoked any number of times. It depends on the load of the server node, and will probably report differing answers when called at different times.

   ii. **Post requisites:** the method does not perform any state-altering operations. It simple queries the server node and finds out the resources required to checkout the customer. It leaves the system in the same state as before the invocation, and therefore has no post requisites.

   iii. **Parameters:** the method uses the parameters of the checkout operation. These are:

      a. **clientid:** the unique identifier of the customer
      b. **items:** a comma-separated list of product items in the customer’s cart
      c. **qrys:** the comma-separated list corresponding to the quantities of the items being checked out

   iv. **Return Value:** the method returns a **boolean** typed value indicating whether the manager can checkout the customer (true) or not (false).

3.2.5.5 **StockManager**

All the stock-related operations are forwarded to the stock manager. It controls what items leave and enter the stock, but its control is restricted to the local stock, and the remote stock operations are requested through the server manager. It is possible to bypass the ORB layer and perform remote stock operations through the JDBC layer, but that reduces the load on the ORB components and makes it harder for us to test the
performance of such technologies under large network loads. The operations that the stock manager performs are listed in the form of public methods in the StockManager interface.

1. pullOut(): the stock manager is responsible for removing items from the stock when they are listed in the customer’s cart. The quantities of these items are checked against the ones in the stock.
   
   i. **Prerequisites:** the method will perform database queries and therefore needs a usable database connection. This is provided by the static methods of the ConnectionManager.

   ii. **Post requisites:** after the method returns the stock of the product item requested is reduced by the quantity requested if initially available, or otherwise by whatever quantity is left in the stock.

   iii. **Parameters:** the method requires the product and the quantity to remove:
         
         a. **product:** the product’s name, which is then used to lookup the stock of that product. This is of a **String** type
         
         b. **quantity:** the number of items of that product to remove from the stock. This is of an **integer** type.

   iv. **Return Value:** the method returns the number of items of the product that have been removed from the stock. This could be less but never greater than the requested number. The number is of an **integer** type.

2. pullOut(): a very useful method for pulling-out more than one item from the stock. This method relies on the previous method – pullOut() – to achieve its functionality. However it is used when a list of products need be removed from the stock in corresponding numbers.
   
   i. **Prerequisites:** There are no prerequisites of this method, because it forwards the work to a different method.

   ii. **Post requisites:** after the method returns the stocks of all the product items in the list of items have been reduced by the quantities requested, based on the functionality of the pullOut() method mentioned earlier.

   iii. **Parameters:** the method requires the list of products and their corresponding list of quantities to remove:
a. *products*: the list of products’ names, which is then parsed and fed into the `pullOut()` method. This is of a *String* array type

b. *quantities*: the numbers of the products to remove from the stock. Again this is parsed in conjunction with the products list, and is fed to the `pullOut()` method. This is of an *String* array type

t. *Return Value*: the method does not return anything, and is thus useful only when the number of removed items is not important.

3. `getProductID()`: this method is a query method used by the stock manager and other classes in the application to find the identifier – from the database – that represents the product given. In turn, this method run a query against the database and returns the requested identifier.

i. *Prerequisites*: There are no prerequisites for this method, because it queries the database and catches any exception that might be thrown. If such an exception is thrown, the method is expected to return a number that could never be used as a product identifier – usually a negative number or zero.

ii. *Post requisites*: Being a query method, it does not perform any change to the data structures in the application or to the underlying database structure.

iii. *Parameters*: the method requires the name of the product for which the identifier is to be queried.

   a. *product*: the name of the product, which is used in the SQL statement to find the identifier. This is of a *String* array type

iv. *Return Value*: the method returns the identifier of the product in the database. This is of an *integer* type.

4. `canPullOut()`: this method is called by the server manager to find out if the stock manager at a certain server node is capable of pulling out the requested item. This is usually done prior to invoking the `pullOut()` method of that stock manager, and while the latter method might not fail catastrophically, it is recommended to make sure the stock manager can pull out the specified item. Another important for this is reducing network load. A stock manager is asked to remove a stock only if it can, and this means that only one stock manager in the cluster removes that item.
i. **Prerequisites:** The method expects a product identifier and the quantity of that product. Since the method queries the local database to find out if the server node can pull out this product, all exceptions thrown by that connection are caught. Only if no exceptions are thrown does the method reply positively. Therefore there are no prerequisites for this method.

ii. **Post requisites:** Being a query method, it does not perform any change to the data structures in the application or to the underlying database structure.

iii. **Parameters:** the method requires the identifier of the product and the quantity of that product to use in the query.
   
   a. **productid:** the identifier of the product, which is used in the SQL statement to check for pull-out availability. This is of a `String` array type
   
   b. **qty:** the quantity of the product being queried. This is of an `integer` type

iv. **Return Value:** the method returns `true` when it can pullout the product from local stock, and `false` otherwise. This is of a `boolean` type.

5. **getPrice():** This method is used by the shop to find out the price of a certain product. The prices of the products are distributed to all the server-nodes, and thus invoking this method on any node should return the same value.

   i. **Prerequisites:** The method expects a product identifier to find its price. Since the method queries the local database to find the price of that product, all exceptions thrown by that connection are caught. Therefore there are no prerequisites for this method.

   ii. **Post requisites:** Being a query method, it does not perform any change to the data structures in the application or to the underlying database structure.

   iii. **Parameters:** the method requires the identifier of the product for which the price is to be reported.

   a. **productid:** the identifier of the product, which is used in the SQL statement to check for pull-out availability. This is of a `String` array type
iv. *Return Value:* the method returns the price of the product if found, and zero otherwise. This is of a *double* type.
3.3 Evaluation Criteria

In this section, we present a framework for evaluating DOC technologies. This framework is based on quantitative and qualitative criteria. These criteria are summarized in Table 1 and are described in the following subsections.

Table 1: Evaluation criteria of ORB technologies

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quantitative</td>
<td></td>
</tr>
<tr>
<td>Speed and Performance</td>
<td>The time it takes a logical layer to complete a unique operation while operating divided by the time required for the same layer to complete the same operation separately</td>
</tr>
<tr>
<td>Scalability</td>
<td>The average of the scale-out factors of all configurations that the application is tested in, weighted by the configuration number</td>
</tr>
<tr>
<td>Memory Management</td>
<td>The process by which a software component performs memory consumption at its various processing layers</td>
</tr>
<tr>
<td>Load balancing</td>
<td>The averaged load proximity that the master server achieves during a given service time</td>
</tr>
<tr>
<td>Fault-tolerance</td>
<td>The ability of a process to recover from a fault that occurs within the same process or at a different node in the cluster</td>
</tr>
<tr>
<td>Fail-over</td>
<td>The ability of a cluster of server nodes to achieve continuity of a service through isolating total or partial server failures while preventing fault propagation through the cluster</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Clustering</td>
<td>The ability to put together a collection of nodes in a networked environment that collaborate to provide one or more services that would be either difficult or impossible to provide through a single-node server</td>
</tr>
<tr>
<td>Qualitative</td>
<td></td>
</tr>
<tr>
<td>Maintainability</td>
<td>The ability to replace any object located at any node within the cluster without affecting the behavior of the other objects in the system in a way that keeps the system capable of servicing its clients at least as reliable as it was before the new object was added</td>
</tr>
<tr>
<td>Testability</td>
<td>The ability to detect the correctness of code inside a component either at runtime or at design time without affecting the work of other components in the system</td>
</tr>
<tr>
<td>Repeatability and Reproducibility</td>
<td>The ease by which an application instance can be reworked in a different environment with varying development parameters</td>
</tr>
<tr>
<td>Ease of learning and Use</td>
<td>The collective time spent in putting the technology into commercial use as part of distributed object software starting from minimal knowledge of the inner workings of that technology but with sufficient understanding of the concepts of distributed object technologies</td>
</tr>
<tr>
<td>-------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Portability and operation system issues</td>
<td>The overall process required to move – in part of as a whole – an application that is successfully operating in a certain environment to the target environment</td>
</tr>
</tbody>
</table>

### 3.3.1 Quantitative Criteria

This category includes the criteria that require experimental evaluation. Each criterion might require a different evaluation method, which in turn might lead us to the development of various experimental procedures to properly evaluate it. In the process of our thesis we realized that some of these criteria do in fact require separate experimental setups, but it was both in our interest and part of the specifications we started with that we create a unified enterprise application to host all of the code required to produce the results.

The various evaluation criteria that we deal with in our study are represented as data resulting from running our enterprise application under pre-defined conditions and for pre-defined periods of time. The resulting log files – as they are generated – are studied and analyzed with the help of a separate application that we developed primarily for this purpose. We call this application the “Indexer”, the reason for this being the way we represent the criteria in our report. We wish to tabulate the criteria along with the technologies studied. Every technology will have a corresponding value for the various criteria in this table. The value is the index of the technology for that criterion. The
indexer will thus take as input the logs files generated from the various runs in the many configurations chosen, and it will generate the tabular representation of the criteria.

The quantitative criteria proposed in this section have the following properties:

- **Criteria Independence**: we should be able to conduct the same experiments using the same enterprise application to study any criterion on its own. Thereby we will require that such criteria have a complete and independent logical structure within the application. This property is the basis of our analysis, and the fact that the criteria can be isolated plays a very important role in generating evaluation reports statistically without the need or reliance on the other evaluations.

- **Data Collectable**: it is required that every criterion have a representation in the form of data in log files. The log files should span the criteria, in that the analysis performed on the log files is both sufficient and required to correctly identify the indexes of the criteria at hand.

- **Stability**: This property allows us to conduct an experiment for prolonged periods of time, yet still be able to report the results within a very small interval of uncertainty. This property goes beyond our particular experimental setup. In any given setup and any given implementation, the criterion should result in the same evaluation within similar intervals of uncertainty.

### 3.3.1.1 Speed and Performance

The primary concern of almost all application users and developers alike is the performance of their application compared to other applications delivering the same content. Users expect the work to be done in a very short time, and a prolonged response time by the server(s) might cause users to give up on the work and abort, reducing interest in the application. But very few realize that speed and performance are vaguely defined. Therefore we need a proper definition of a "fast" application or a "fast" technology before we can correctly evaluate this criterion.

Whether distributed or running on a single-node machine, the speed an application is defined by the time it takes to process and complete a transaction. A very common unit of measurement is used in this context. The number of transactions per
second – TPS – is sometimes used interchangeably with the application speed. But there
is more to speed than transactions time. An application is composed of several logical
layers, and the TPS number of the application does not reflect the “speed” of any of the
constituting layers. The TPS number gives an overall averaged evaluation of an
application performing transaction in statistically accepted numbers. A very common
example of such an application is a database server.

As for performance, we will adopt the following definition to assist us in
evaluating the performance the various application(s) we are dealing with. We can speak
of performance only for logical software layers. The performance of an application can
be said to be the average of the performance of all the logical software layers that it
comprises. Thereby, we define performance of a logical software layer as the time it takes
this logical layer to complete a unique operation while operating as part of the entire
application, divided by the time required for the same logical software layer to complete
this unique operation when operating separately. This definition might seem a bit
overstated, but when thought of in terms of modern multithreaded distributed enterprise
applications, it does give a better measure than the classical TPS. Our definition bears
great similarities to the BBPM performance measurement unit. The Business Blocks Per
Minute definition states that an application’s performance is defined by the number of
“business blocks” it can compute or process within a minute of time. It is possible to
move from our unit of measurement to BBPM rather straightforwardly. By adopting our
definition we will therefore require that the performance all of the logical units making
up the enterprise application be evaluated separately, and afterwards combined to form
the performance index of the application.

Let $T^L_S$ be the time required to execute a business operation at the $L^{th}$ tier of a
DOC application in a single node setup, and let $T^{LC}_D$ be the time required to execute the
same business operation when a configuration $C$ is setup in a network environment. Let
$N_L$ be the number of layers in the DOC application, and let $N_C$ be the number of
configurations that the DOC application is run under. If we denote by $\alpha$ the index of
performance of the DOC application, and if we denote by $\alpha_T$ the index of performance of
the ORB technology $T$ used by the DOC application, then:
\[ \alpha_1 = \frac{T_D^{lc}}{T_s} \]  

\[ \alpha_2 = \frac{\sum_{i=1}^{N_i} \alpha_i}{N_L} \]  

\[ \alpha = 1 - \frac{\sum_{c=1}^{N_c} \alpha_2}{N_C} \]  

\[ \alpha_T = \Theta(\alpha) \]  

(1a)  

(1b)  

(1c)  

(1)

3.3.1.2 Scalability

Scalability is a well-known property of all distributed application. Many formal definitions of scalability can be found in the literature, but the one that we will adopt is based on the various configurations we tried our application in. But before we arrive at this definition, we will define another quantity that we call the scale-out factor. We speak of the scale-out factor in terms of the number of server nodes that we use in our experimental configurations. For example when we use 5 nodes as server nodes, we have (as regards our application) 4 configurations that we can run the application in, excluding the single-server configuration. Therefore the scale-out factor is the number of times the application is run successfully when moved from an \( n \)-server cluster to an \( n+1 \)-server cluster, divided by the total number of times that application is run. The server nodes not being used are employed as client nodes, making the factor differ with the number of nodes making up the cluster. The success or failure of a run is subject to the application itself.

With this definition of the scale-out factor, we arrive at our definition of the scalability of an application. Obviously this is the average of the scale-out factors of all configurations that the application is tested in, weighted by the configuration number.

The scalability of a distributed application gives us an idea of how easy or difficult we can add more server nodes to our cluster. Many distributed applications are intrinsically incapable of operating outside a certain cluster size. The scalability of such an application helps the company or institute that wishes to use this application in deciding on the best or most acceptable application scalability it finds suitable for its
needs. An application might have a high scale-out factor for large cluster sizes, but its use might be of most interest in small-sized clusters, rendering it useless in such environments.

Let $N^C_S$ be the number of server nodes used when the DOC application is run in configuration $C$ of a network environment, let $N^C_T$ be the total number of server and client nodes used in configuration $C$, and let $N_C$ be the number of such configurations. Let $\beta_T$ indicate the scalability index of the ORB technology $T$ used by the DOC application. Then:

$$\beta_i = \frac{N^C_S}{N^C_T}$$  

$$\beta_T = \sum_{C=1}^{N_C} \beta_i$$  

(2a)

(2)

3.3.1.3 Memory Management

Among the basic requirements of every enterprise application is staying functional for prolonged periods of time. This requirement is the reason why such applications are used as servers. But if an application is to run all the time, we should make sure that servicing clients is done in a memory-efficient way. Even the largest repository of memory cannot last when an application undergoes memory leaks or abuses the memory allocated to it. Therefore in such environments and under such circumstances we lose the memory luxury that many desktop applications enjoy.

The management of memory is different from the consumption of memory. But in the case of a distributed application, the application and the ORB perform memory consumption, while memory management is done at the ORB level. This carries inherent complications when trying to compute a memory management index for the application in its various ORB implementations. The question is how do we compute the management index of a middleware component, a software layer that we have no control over. We can easily compute the memory consumption index – as we explain in the next paragraph, but how do we go about computing the consumption index when no application logic is involved. Clearly we cannot factor out any of our business logic, and
we cannot insert business logic into the ORB middleware. Before we can find a solution we need to define what memory management is, and how it can be identified in a distributed environment.

Memory management is the process by which a software component performs memory consumption at its various processing layers. This means that the software component should have a mechanism by which it requests memory. In our efforts to find out the memory management index of the different ORB under study, we do not with to go deep into the programming language memory management algorithms. We will assume that if a technology – which is a software component after all – uses different memory management algorithms than those used by the programming language in which it is implemented, then this technology is eligible for memory management indexing. As is obvious from the above statement, we require access to the algorithms of the technology, something we do not have. We find ourselves in the situation where we have to draw the following conclusion: ‘A memory management index cannot be computed for the ORB technologies under study’. The memory management algorithms are located in the programming language subroutines that the ORB technologies are implemented in. Since we do not have access to those algorithms – neither is it possible to go into those algorithms had they been available – we can only speak of memory consumption of a distributed application using a particular ORB technology for achieving object communication.

To compute the memory consumption of our enterprise application, we will follow a similar approach to that used in measuring performance. We will record the total number of bytes of free memory before and after every business transaction that our application performs. This is to be performed at every server node, and we will weigh this measure with the number of business logic operations performed at the respective server nodes, and finally average over the number of server nodes.

Finally, the memory index – as we shall call it – represents the total memory used by the application and the ORB technology. The same application logic is present in all versions of our application, and if we indicate as $M_{ORB}$ the memory index of the ORB technology at hand, and $M_{APP}$ as that of the application, then we can say that $M_{ORB} = \Theta(M_{APP})$. 
Let $B^{OC}_D$ be the number of bytes of memory used by the DOC application when processing a business operation O within a configuration C in a network environment, and let $B^{O}_S$ be the number of bytes of memory used by the DOC application when processing the same business operation O in a single node configuration. Let $N_C$ be the number of configurations that the DOC application is executed in, and let $N_O$ be the total number of business operations used. If we denote by $\gamma_T$ the memory management index of the ORB technology T, and by $\gamma_0$ the memory management index of the DOC application, then:

$$\gamma_1 = \frac{B^{OC}_D}{B^{O}_S}$$

$$\gamma_2 = \frac{\sum_{C=1}^{N_C} \gamma_1}{N_C}$$

$$\gamma = 1 - \frac{\sum_{o=1}^{N_O} \gamma_2}{N_O}$$

$$\gamma_T = \Theta(\gamma)$$

(3)

3.3.1.4 Load Balancing

The purpose of building distributed applications is to distribute the load from a single-node to a multi-node framework. The application components would thus be located at different server-nodes, and communication between their constituting objects would be handled through an ORB middleware. Application logic would have to be distributed as well, and it is this distribution that dictates object distribution. In such an environment, the application needs to have a central server where client interfacing is conducted. During the course of our work, we found out that this paradigm is much better than a distributed client interface. We found that it is less troublesome to have a main server – which we call the master server – interface all the clients and forward their services to the available server nodes than having the clients communicate directly with the servers. This structure allows us to “hide” the slave server nodes – as we have come to call them – behind one master server. Adding more servers to the network is now
easily done, because the clients would not notice the new servers, since they do not communicate with them in the first place.

From here come the ideas of load balancing and clustering. We will talk about load balancing now, and discuss clustering in a subsequent section. It is in the interest of the master server to forward client requests to available server nodes. This not only reduces the load on the master server, thus allowing it to handle more clients, but it also increases server response time, resulting in an increase in overall application performance. The reasonable load-distribution algorithm to use is forwarding a particular client request to the least busy server. But since this is a very dynamic environment, the master server should continually update its server-load table to keep track of server loads as the servers complete their work. If a server is very busy at a particular instance, it will no receive client requests, thus giving it time to finish handling the requests it has. This would free this server and it should be given more clients to handle at a later stage. The primary goal of the master server is making sure the servers have comparable loads at various intervals of time. Thus we define the load-balancing index as the averaged load proximity that the master server achieves during a given service time.

Let $\mathcal{N}_s^{oc}$ be the number of business operations forwarded to server $S$ by the master server when the DOC application is run in configuration $C$ in a network environment. Let $\mathcal{N}_o^{c}$ be the total number of business operations performed by the DOC application in configuration $C$, let $\mathcal{N}_s^{c}$ be the total number of servers used in the same configuration $C$, and let $\mathcal{N}_c$ be the number of configurations used. If we denote by $\theta_1$ the load balancing index of the ORB technology $T$, and by $\theta$ the load balancing index of the DOC application, then:

$$\theta_1 = \frac{\mathcal{N}_s^{oc} - \mathcal{N}_o^{c}}{\mathcal{N}_s^{c}}$$

$$\theta_2 = \frac{\sum_{S=1}^{N_s^{c}} \theta_1}{N_s^{c}}$$

(4a)  

(4b)
\[
\theta = \frac{\sum_{C=1}^{N_C} \theta_C}{N_C} \\
\tau = \phi(\theta)
\]

3.3.1.5 Fault-Tolerance

The ability to tolerate failures has always been a concern when designing mission-critical applications. The term “fault-tolerance” has been used to describe the attributes of such applications. Before we discuss fault-tolerance in our framework and how it is affected by the various ORB technologies used, let us discuss fault-tolerance in both the single-node and distributed environments.

It is a well-known fact that along with the growth of any software component or application, the probability of something wrong occurring increases. The term “failure” is used to describe the failures that “can” be avoided — but might not necessarily be avoided — in the software. On the other hand, “error” is the term used to describe those failures that — once they occur — cannot be detected or avoided without causing the entire application or software to fail irreparably.

But with the growth and maturity of application logic, and with the increase in software layers within a single application, these terms started to take on different meanings. It is now necessary to identify in what software layer the error or fault has occurred, and whether the failure affects the layer itself or the layers above and underneath that layer. In the case of a distributed application the failure can occur in a particular layer at a particular node, and it might not be noticeable by other nodes in the cluster. Therefore, it is important to properly define what fault-tolerance and how it is to be implemented in a distributed application model. We found that fault-tolerance requires a great deal of work to implement in its classical sense, and thus we had to arrive at a more “distributed-oriented” definition.

Rather than defining fault-tolerance from the perspective of the software layers making up the distributed application, we chose to define it within the distributed context. But before defining fault-tolerance, we had to arrive at a definition of the term “fault” and what it means in the clustered, load-balanced environment that our application is working
in. We define a fault as the “inability of a server-node to process a request that another concurrently-active server-node in the same cluster would otherwise complete successfully within the constraints set by the requesting node”. To prove that this definition of a fault is acceptable in a distributed environment like the one we are working with, it has to default to the classical definition when the distributed constraints are removed. When the application is running on a single-node, the request becomes identical to a system call or a method invocation. And while the server-node becomes the application itself, the “other” concurrently active node becomes a different instance of the same application running on the same or another machine and processing the same workload. The definition of a fault is thus the “inability of an application to respond to an invocation which it would otherwise succeed to respond to had it been performed in a similar environment”. The constraints that our definition mentions might include a time-out set by the requesting client, and when it does not receive a reply within this time it assumes that the server had failed. We identify the client in a single-node application as a software layer in the application itself.

When such a definition is adopted in our distributed framework, it becomes clear that an “error” is not identifiable. We exclude errors from a distributed environment, and identify them with the definition of the “fault” that we presented. This approach is backed up by the implementations of the various middleware that we study, and the nature of the Java programming language. In Java, an “error” and an “exception” – the term used to identify a fault in the Java lingua - are similarly handled, and ORB implementations in Java define exceptions and errors alike, leaving it to the application to decide on the nature of the fault.

Having adopted a definition for the term “fault” in the distributed context, we realized that there is no clear definition for the term “fault-tolerance”. When a fault occurs in a distributed environment, it occurs within any one of the following layers:

- **The System Layer**: the fault could have propagated to the application layer due to a failure in a system component. Since the system layer is the lowest in the hierarchy, we expect such a fault to propagated to the application layer – the highest in the hierarchy – if the ORB layer does not handle it properly.
• The ORB Layer: this layer is responsible for the propagation of information from and to the existing nodes in the cluster. In the case of faults, the information – which is now faulty – either continues to propagate, which means that other nodes in the cluster will notice the fault’s occurrence, or it does not propagate at all, putting the faulty node in an inactive state, thus not affecting the application as a whole.

• The Application Layer: this is where the business logic resides. In case of an ORB fault the application should use ORB-specific calls to determine the cause of the fault. If it cannot do that, then the fault is of a system nature, or it could be due to a failure that the ORB could not identify. Sometimes the fault may be “tolerable”, which leaves the ORB functioning normally, but may incur some damage to specific requests.

The only layer that we can control is the application layer, and thus the fault-tolerance algorithms reside in that layer. It is important to note that the only faults that can be handled are those propagated through the ORB layer, in addition to the faults – both logical and programmatic – that the application itself generates. The procedure of handling these faults differs according to the nature of the fault. If the fault cannot be isolated and the request correctly processed, we talk of a non-recoverable fault. However, if the application can isolate the fault and identify it with a particular logical block, then the application may be able to work around the faulty request and therefore process the request correctly. In such a situation we talk of recoverable faults. It should be noted that both situations are within the fault-tolerance of the application. Therefore the application is expected to survive the faults, no matter what their nature is.

Fault recovery is therefore implemented likewise in the three layers aforementioned. In the case of a system fault, the underlying operating system is expected to recover from the fault and therefore isolate it and prevent its propagation to the ORB layer. The operating systems that we use to conduct our experimentations are expected to be fault tolerant. Not all faults are hidden from the ORB layer, and not all ORB layers expect the underlying operating system to be fault tolerant. Being written in
the Java language, such ORB implementations rely on the Java routines, and these may or may not fail depending on the operating system.

The application layer fault-recovery techniques are the ones we find most interesting. When the application encounters a fault, it should identify the fault's origin and act accordingly. There are many techniques to recover from a fault at the application layer, but in our situation we will be dealing with three types of fault-recovery techniques:

- **Node-specific**: in this technique the fault is identified with a particular node in the cluster. The node fails to communicate with the master server that forwards to it the client requests. The algorithm used by master server includes continuous server node detection. Should a node fail to report its current workload, the master server stops forwarding future requests to this node, but does not halt the current requests that the node is processing. If the node later resumes communication, it either returns the results of the requests if there are any pending, or otherwise joins the active nodes in the cluster.

- **Cluster-specific**: when the entire cluster of server nodes fails, this means that the network is faulty or the ORB is experiencing communication problems. The master server follows the same algorithm that it uses with node-specific faults. Since the entire cluster is faulty, the server cannot process any requests in the absence of working nodes, and thus it will reject further requests. While the master server could handle the requests, it does not do that for a very specific reason. Should the cluster be accessible at a later time, the master server should be ready to forward the requests to the active nodes. If the server is busy processing client requests instead — and such requests might cripple the master server — it will not be able to perform it functions properly.

- **Master-Node-specific**: the entire cluster is managed through a master server-node that performs all communication operations required to process client requests. In addition to these operations, the master node is responsible for load balancing and other server-oriented tasks. In all, the master node is the strongest communication point in the cluster, and thus should be kept alive for the application to run. Some fail-over techniques have been implemented to take over the master ownership.
when that server fails, but it is best that the master server do the work from start. Since the application logic states that the master server should do very little other than synchronization and collaboration in the cluster, the probability of a master fault is very small. It should be noted that we are dealing here with application logic faults, or faults propagating from the ORB. If ORB faults exist, the master server will simply handle them and abort any process that the fault affects. This is important and critical, since any effort spent to recover faulty processes may reduce server management efficiency. During the course of our work we have found it very rare that an application-specific fault may exist at the master server node.

Let $R_C^S$ be the number of faults recovered from by the DOC application at server S when the application is in configuration C in a network environment, and let $F_C^S$ be the total number of faults that occurred at server S in the same configuration C. Let $N_C$ be the total number of configurations, and let $N_C^C$ be the number of servers used when the DOC application is in configuration C. If we denote by $\psi$ the fault tolerance index of the DOC application, and by $\psi_T$ the fault tolerance index of the ORB technology T used by the DOC application, then:

\[
\psi_T = \frac{R_C^S}{F_C^S} \quad \text{.......................... (5a)}
\]

\[
\psi_2 = \frac{\sum_{C=1}^{N_C^C} \psi_1}{N_C^C} \quad \text{.......................... (5b)}
\]

\[
\psi = \frac{\sum_{C=1}^{N_C} \psi_2}{N_C} \quad \text{.......................... (5c)}
\]

3.3.1.6 Fail-Over

One of the criteria that we try to evaluate in our enterprise-distributed application is the ability of a server node to fail-over to another server node. The concept of fail-over
is useful when targeting high uptime percentages in dedicate servers. In a server cluster, if a server node fails for any reason, the cluster can hide this failure from directly connected clients by dedicating a stand-by server that detects this failure and "impersonates" the failing server. The failure is not visible for the client, and in fact the client does not notice any irregularities.

Many factors are involved when it comes to fail-over implementation in a distributed cluster. First there is the fail-over time. This is defined as the time needed by the standby server to completely regain control of the failed server. The more queries the standby server makes to the targeted server, the more the network is used, thus reducing performance and affecting communication. On the other hand, the fail-over time is required to be very small, requiring efficient algorithms.

The second factor is the locality of the fail-over implementation. It is possible to implement fail-over very simply through DNS configurations. If a dedicated server node is assigned to constantly query server nodes, it can detect failure and modify the entries in the DNS server on the network to guide client requests to an alternate stand-by server. The same thing can be done without the need for a DNS server. Through "naming" services implemented in the ORB middleware, a standby server can query the naming service to detect an ORB failure at the target node. It then registers itself in place of that failing server, and the requests are thus transferred to it. Here the difference in approach reflects on the way requests are handled. If external fail-over is used, then the requests may not get interrupted and the client does not detect server interruption. Some requests may get corrupted, but are not interrupted. If, however, cluster fail-over is implemented, then the requests may fail but the service is very soon resumed. If no clients are connected at the failure time, no information is lost.

It is important in this context to note that the "client" may be a server-node in the cluster itself. Distributed applications that employ server-to-server communication, fail-over plays a vital role is keeping the central – or master – server alive and thus the entire cluster working. This is the case in our application.

We thus define fail-over as follows: "It is the ability of a cluster of server nodes to achieve continuity of a service through isolating total or partial server failures while preventing fault propagation through the cluster". The definition sets fail-over apart from
fault-tolerance through the condition of "fault-propagation prevention". In a fault-tolerant application the fault is propagated but otherwise handled and preferably recovered from. In the context of fail-over, the fault is isolated within two or at most three server nodes in the cluster. A successful fail-over takes very little time, and before the other server nodes in the cluster can detect this failure or react to it.

The implementation of fail-over in a distributed cluster depends on the type of fail-over adopted. Partial fail-over is the easiest to implement, requiring only a standby server node that monitors the target server. If a naming service is present, the standby server will have to register itself once a failure occurs. When the target server regains functionality in the cluster, it is better to avoid a retake by that server in order to reduce cluster instability. In our case we adopt partial fail-over because it is the cheapest approach when it comes to network and master-server loading.

Total fail-over – which we also call uninterrupted cluster servicing – requires duplicating all operations at the standby server. This means that every process that the target server performs is replicated at the standby server. It is also advised to treat the standby server as if it were a master server itself. It should query other server-nodes in the cluster, thereby maintaining a load-table similar to the one at the target server. The only difference is the name associated with the standby server at the naming server. When a failure occurs at the master server, the naming service swaps the names of the target server and the standby server. Client requests will be uninterrupted, and the cluster will not undergo any noticeable instability. Later when the target server regains cluster membership, the naming service treats it as a standby server, and the roles are swapped accordingly. In this context it is possible to duplicate standby servers as desired, but care should be taken because this incurs additional network loading.

Let $S_C$ be the number of successful fail-over events forced on the DOC application when running in configuration $C$ of the network environment, and let $F_C$ be the total number of fail-over attempts in the same configuration $C$. Let $R_C$ be the average of the response times of the DOC application in configuration $C$ when successfully failing over. When $R_C$ is computed it should be normalized, such that the smallest value possible is greater than 1. The response time is defined as the time needed to forward all business operations to the standby server when a fail-over is attempted. Let $N_C$ be the
number of network configurations that the DOC application is tested in. If we denote by \( \omega \) the fail over index of the DOC application, and by \( \omega_r \) the fail over index of the ORB technology T that the DOC application is using, then:

\[
\omega_i = \frac{S_c}{F_c} \left( \frac{1}{R_c} \right) \sum_{i=1}^{N_c} \omega_i \tag{6a}
\]

\[
\omega = \frac{\sum_{i=1}^{N_c} \omega_i}{N_c} \tag{6b}
\]

\[
\omega_r = \Theta(\omega) \tag{6}
\]

### 3.3.1.7 Clustering

The term “cluster” is used to identify a collection of nodes in a networked environment that collaborate together to provide one or more services that would be either difficult to provide through a single-node server or otherwise impossible to accomplish. There are many forms of clustered nodes, and many ways exist to achieve clustering. We will discuss the procedures we used to implement distributive clustering using the various ORB technologies under study. We will also talk about clustering in general, and propose ways to implement its various types and forms.

During the course of our study, we were able identify the following types of clustered nodes:

- **Distributive Cluster**: this is where a service is logically divided into separate parts that can be performed at different locations within a group of nodes. There should exist a single entry node to the cluster that interacts with the outside. All requests and are received and transmitted through this node. Here it is highly recommended to implement fail-over at the entry node, or otherwise failure of that node will put the cluster in a state of “blackout”.

- **Synchronized Cluster**: in this type of clustering, all the nodes are capable of performing the required services independently. However the states of these server nodes are kept “similar” at all times through a centralized synchronization server. This node makes sure that all the nodes in the cluster have access to the same or similar resources at all times – which could include a database server or a
network storage. Clients connecting to the cluster will experience the same servicing conditions regardless of the server node providing the requested service. 

- *Parallel Cluster:* This is similar to a distributive cluster, but instead of distributing the workload in a logical way, the master server node subdivides the request into similar and complementary parts, each performed at a different server node, and the result is later joined at the master server. Dynamic clustering is achieved through run-time parallelism, thereby delaying distribution of the tasks until the master server determines the number of nodes available.

We have adopted the distributive clustering model, and thus we used server nodes that are orchestrated through a central master server. In addition to load balancing in the cluster, we also implemented fail-over at the master node. The index of the cluster is simply the ratio of the number of server nodes participating in the cluster and the number of total server nodes available.

Let $N_{CN}^S$ be the number of servers used in a configuration $C$ of a network environment $N$, and let $N_{CN}^N$ be the total number of nodes available in the same configuration $C$ of the same network environment $N$. Let $N_C$ be the number of configurations in which the DOC application is run, and let $N_N$ be the number of network environments that the experiment is conducted in. Since a configuration involves fewer nodes than are available in the network, $N_C$ is less than $N_N$. If we denote by $\xi$ the clustering index of the DOC application, and by $\xi_T$ the clustering index of the ORB technology $T$ used by the DOC application, then:

$$\xi_1 = \frac{N_{CN}^S}{N}$$

$$\xi_2 = \frac{\sum_{C=1}^N \xi_1}{N_C}$$

$$\xi = \frac{\sum_{N=1}^\infty \xi_2}{N_N}$$

$$\xi_T = \Theta(\xi)$$

\[ (7a, 7b, 7c, 7d) \]
3.3.2 Qualitative Criteria

The category of qualitative evaluation criteria that we attempt to evaluate in this thesis includes a number of criteria that we found both useful and important when studying distributed object software. We were not able to evaluate these criteria through an experimental setup similar to the one we used for the criteria in the empirical category because of the nature of the qualitative criteria. These criteria cannot be studied experimentally, but rather evaluated through the entire course of working with distributed software. As will become clear later, the indexes of these criteria were based on our subjective evaluation. We felt we could provide an independent overview of the various ORB technologies we used, and in every context that we used them we tried to find a comparative value for the criteria in this category.

The list of the criteria is not exhaustive. We feel there are more criteria that can be added to this category, but we advise that the criteria be fitted first into the empirical category through a proper setup, and only if this fails should they be added to this category. Some of the criteria already in this category may turn out to be empirical in nature, and careful considerations might result in setting up the proper experiment that can measure or evaluate such criterion. However the time and generality constraints that we had in our study made us decide on putting these criteria in the qualitative category.

The following is a list of these criteria. We will try to explain what they are and try to give them a proper definition adopted from the distributive nature of the work we did:

3.3.2.1 Maintainability

A very important factor in the success of any application is the ability to support the software after it has been put to use. Many requirements come up when the application becomes functional, and real-life testing and usability of the software may incur new features that were unseen when the application was first developed. If the application design affords late-time additions to the system, then the application as a whole becomes easily maintainable. It is very important that the application be built from ground up with maintainability in mind. Component-based modeling and other object-
oriented design approaches simplify maintainability, since errors or additions can now be localized within one or more components and layers in the system, and these can be either replaced or added at any time.

But first let us properly define maintainability in the context of a distributed application. Distributed object software is maintainable if “any object located at any node within the cluster can be replaced or otherwise added to the system, without affecting the behavior of the other objects in the system in a way that keeps the system capable of servicing its clients at least as reliable as it was before the new object was added”. The definition defaults to the single-node definition. Through maintainability at the single-node, we expect the services of the application to continue being provided.

When working with distributed object software we encounter many different situations where maintainability is required. We will list these types and situations and suggest work methodologies and practices based on our experience with such software.

- **Functional Maintainability**: the simplest of these types is replacing functioning software with a different version that includes new features or corrected ones. This is also known as “updating” the software, and is a common practice in all software-engineered applications. In a distributed environment, the new features should be broadcast to all the nodes in the cluster. While the simplest way to do that is to replace the entire software with the updated one, in a distributed environment with applications running all the time and in a remote area with minimal technical support, this becomes a very risk-involving procedure. Most programming languages implement run-time functionality detection, through which a component's operations are exposed to the other components of the application when that component is used. Examples of such components are Java beans and COM objects. Making use of these programming language features in such distributed applications makes the job of detecting new component operations less tedious.

- **ORB Maintainability**: sometimes it is required that the application performs certain operations that cannot be implemented directly in the application but rather require the existence of features in the underlying ORB. While function maintainability is possible due to the direct access to the software components,
ORB maintainability is not always possible or desirable, since tampering with the ORB – if possible – involves a thorough understanding of the ORB components that need be changed. Many if not all applications developed using an ORB middleware do not require such knowledge on the part of their developers. While almost all middleware used to perform distributed computations share a general common programming paradigm, differences exist that make using one ORB more desirable than the other, and when this decision is taken after the application becomes functional, the job becomes very difficult if not impossible at times. We dealt with such a situation when we developed our enterprise application because we had to come up with as many versions of the application as we had ORB to study. Since all the versions need to perform the same basic operations, our application development defaulted to an ORB maintenance procedure. Therefore we realized the importance of separating distributed operations – sometimes within the same component – from other system operations, and sorting out the ones that are common and sharing them across all versions. A similar approach should be followed every time a distributed application is developed using an ORB middleware. While it is very unlikely that ORB maintenance might be required, it does improve the application design and quality of service.

- **Environment Maintainability**: this is the most difficult and most occurring of all the maintainability types. Since distributed applications run on the operating systems communicating through a physical network, changes to that network or to the operating systems might reflect on the application during its course of work, and this calls for environment maintainability. Examples of such changes are upgrading the network cabling to support faster communication. If the application was tuned to operate at certain network speeds, some communication tasks might gain faster pace and thus affect the application’s services. Another example is scaling the application. As discussed in the scalability criterion section, some distributed applications have a certain scaling factor at which they operate best. If the distributed environment is later upgraded to include more or less nodes, the application might lose some of its features. Given the complexity of such
environments and the numerous possible configurations, it is prohibitively
difficult to propose a general procedure or approach to this type of
maintainability. It is recommended, however, that the target environment be
carefully studied before the design stage, and the heuristics of such an
environment be carefully extrapolated, in hope of finding the most probably
environment changes and adjusting the application correspondingly.

Finally, the maintainability index can be measured in a distributed application by
proposing a sequence of updates or changes to the application after the design stages have
been completed. If the application was designed in such a way that distributed operations’
code is separated from other common operations, then the index is the ratio of the
successful implementations of these changes to the total number of changed requested. If
computing such an index is possible then the criterion is more of an empirical one. We
will not perform such a computation, but rather assign a subjective index value based on
our overall experience.

3.3.2.2 Testability

Testing is one of the most important steps in the lifecycle of a software
application. We cannot stress enough the importance of employing effective testing
procedures during the entire course of the software lifecycle. While a thorough discussion
of testing techniques falls beyond the scope of our thesis, we will discuss the various
testing techniques that we found useful and necessary when developing a distributed
application.

Testing a single-node application can be a very complicated process. Indeed it is
deemed complicated enough that a dedicated team is assigned to escort the development
stages of such applications. The usage of test cases is very common, and the application
is subjected to various conditions that might arise in a real-life situation. Thorough and
complete testing is not possible, and therefore development teams often find themselves
minimizing the possibilities of things going wrong. Special tools have been developed to
assist in testing, and such computer-aided testing tools are even integrated into the
development tools that are used.
Testing a distributed application involves all the familiar difficulties of its single-node counterpart, in addition to the newly added problems faced by a distributed application. Not only is the application the target of testing, and rather than dealing with one source of potential failures, additional players are involved that make it far more challenging to test a distributed application. These might include the environment within which the application is to operate, the middleware that the application uses, and the various deployment techniques that the application is to be installed through, in addition to many hidden agents such as the number of users and the cluster size. We will try to organize these testable targets, and link them to the application that we developed and to our experience with distributed object software. Later we will try to come up with a way to measure testability of a distributed application and its underlying ORB middleware.

The targets of testing in a distributed environment include the following:

- **Application Components**: Distributed object software is primarily made up of components distributed across various cluster nodes. This is the case of object-oriented or component-based software, and the components that make up the application are the primary testable targets. In this case, classical single-node testing techniques should be used to make sure the components behave properly under various working conditions. Since it should be possible to convert or default every distributed application to its single-node version, component-testing techniques should be employed and preferably prior to distributing the components over the cluster nodes. If no single-node version can be realized – as is the case in some custom-built distributed applications – it is advised to create special simple stand-alone applications that use these components – separately if possible – and test the applications using the single-node techniques.

- **Communication Components**: The primary source of communication logic, these components or functions should be tested separately to isolate potential communication problematic code segments. Testing should include a similar networked environment to that of the deployment target. In addition, the testing methodology should be as general as can be, to facilitate replicating the test cases over all possible configurations of the network. The easiest approach is to come up with simplified distributed applications that use these components. When
testing such components and any other components, it is important to follow and object-oriented programming approach. Test components are desirable because later they can be plugged into any application that uses these components. Unlike the application components, communication components are usually reused across many distributed applications, so the survivability of these components is very important. The same applies to the testing components, and once these are completed then a distributed application becomes simple to build and easy to test.

- **System Components:** The underlying operating system uses certain components—such as dynamic link libraries in the Windows platforms—to provide services to the applications running underneath it. Testing these components is not as possible as the other components in the application, and sometimes they are taken for granted to work. In the section about environment maintainability—see the maintainability criterion—the choice of environment, including the operating system, might change later when the application is put to use. In such situations, and with the continuous growth of operating systems, some components might change that render the application incompatible. While testing these components is usually not possible, it is advised to use programming languages that are require minimal changes to the code when porting across operating systems. Java is a perfect example, and instead of testing the components of the operating system, the core components of the language can now be easily tested—and replaced when needed. This is one of the reasons why we used Java as our language of choice. Whenever an operating system becomes inappropriate, we can easily port to a different platform with minimal changes to the application code. The components in the Java library can be replaced with custom built components, and we can easily plug in our own test components.

As can be noted from the components listed above, they reside in the layers that make up the distributed application. Such component testing may be referred to as layer testing, in which the layers are tested, and later joined together and tested through the familiar system testing techniques. While we did not mention the exact details of the
testing procedures for these components, it is understood that each requires a different setup and set of tools to accomplish the desired results.

3.3.2.3 Repeatability and Reproducibility

Very often, during the course of building distributed object software, we require certain components or methods of work that we have either built before or otherwise are available through third-party components. A very important principle of software engineering is the ability to reproduce certain work with minimal ease, localizing all the new work within the business logic components rather than doing the system component work all over again.

If distributed software is built using the methodologies we mentioned so far, then making use of single-node components in various layers of the application becomes possible. But there are code segments within the application that cannot be reused, possibly because of their inherent complications or due to their application-dependability. Such code cannot be put elsewhere within any other distributed application, and therefore every time a similar but incompatible situation occurs, the code has to be recomposed and tested and configured from ground-up. The code is thereby labeled “repeatable” and the process of reading it off a specific application and porting it into another is called “repeatability” or “reproducibility”. In the distributed paradigm, such situations are encountered more often, mostly because of the nature of ORB programming and the environment specifics that accompany such software. ORB repeatability is therefore defined as the “ease through which ORB-specific code-segments can be re-introduced into a different distributed application to provide similar services”. The term “ease” makes this criterion incompatible with an empirical evaluation approach. In this thesis we will assign values for the index of repeatability based on our experience with such technologies.

We can point out some situations in which repeatability plays an important role. While not all of these situations are commonly encountered, listing them serves the purpose of clarifying the many complications associated with developing distributed applications:
• *Protocol Repeatability:* using communication protocols abstractedly enhances the design of a distributed application. It is a common practice to build components that use messaging to interact with each other in a networked environment. With the introduction of object broker technology, many old message-based components were replaced with direct method invocation. But as the scale of a distributed application grew, it was necessary to standardize communication within a certain framework. Communication protocols that are built inside the invoked methods play an important role in reducing coding and placing the business logic within configuration data sources that can be edited or changed without the need to stop the application. The code that drives protocol processing is custom-built for the application at hand. To reduce code rewriting – but still not eliminate it – a technique can be used which we will call protocol layering, something similar to application layering. Any protocol built for use in a distributed application should be split into an application-specific layer, holding much of the messaging architecture for the application to deliver its services, and a protocol-management layer that is common to all applications. An example of the protocol-management layer is authentication, which does not change even if work request does.

• *Distributed Configuration:* Any application requires certain pre-assigned values as configuration options to be used while the application is running. The exact location of certain files, the login to use to access a database, or the number of clients to accept, all these serve as examples of information that in a single-node application would be stored in local configuration files. When moving to a distributed framework, such configuration options become themselves distributed. While some information is common to all server-nodes in the cluster, other information is node-specific. Coming up with a proper way of serving these configuration options to the many components in the application becomes problematic as the size of the cluster increases. To handle this problem, special "configurations nodes" are setup to serve these options. The best way to do this is to create a local configuration server – local to every node – and a central configuration server – accessible to all the nodes in the cluster. When certain
configuration options are requested by the application at a cluster node, the local server is queried, which forwards the query to the central one in case it cannot supply the options. Code written for configuration servers is another example of repeatable code. In order to minimize the effort of porting such code, it is recommended to categorize such configuration options in such a way that an option can be located through its category rather than searched for in the entire repository. Special API – application programming interfaces – should be devised to query the configuration servers, which can be used in other applications if needed. This makes it easier to use the same API calls rather than writing them all over again.

3.3.2.4 Ease of Learning and Use

Important to promoting the use of any technology is the ease through which it is introduced into any development environment. Indeed this can be of a greater importance than the actual properties of that technology. The decision of adopting a technology for use in a specific project may very much be affected by the cost required to make this technology work in the deployment environment. If technology makers can reduce the effort required to adapt a developer’s way of thinking to their proposed framework, then it is possible for their products to gain the advantage when market-driven project-management decides on such technology.

As can be concluded for the technical introduction to our thesis, all of the components of the ORB technologies that we experiment with are freely available, either as part of other technologies – such as Java RMI – or in a downloadable format – such as DCOM. However, it is the ease through which distributed applications can be built using such technologies that given them their respective attributes. Therefore in order to decide on what value each technology has when it comes to the ease of learning criterion, we need to properly define, in a distributive context, what the ease of learning of ORB technologies means.

Ease of learning and use of an ORB technology is defined as the “collective time spent in putting the technology into commercial use as part of distributed object software
starting from minimal knowledge of the inner workings of that technology but with sufficient understanding of the concepts of distributed object technologies”. This definition suggests an evaluation approach for this criterion. By collective time, we mean the time spent by the entire development and deployment teams, adding up the individual times of all team members.

Development time is divided into learning time and implementation time. Since the definition states that the individuals using the technology should have sufficient understanding of the concepts of a distributed application, the time that is spent learning is equal to the time spent on resolving the dependencies between the various components of the technology. Therefore we conclude from this and our experience with such technologies the following key properties that help in differentiating technologies of the same type – such as distributed object technologies:

- **Structural Compactness**: every technology uses a considerable number of inter-operating components to function. These components depend on each other, and they use each other in a structural model that makes the technology useful. Such a model of no interest to the user of this technology – although in some situations it is – because what is required is to fulfill the operations needed by the application. However some technologies rely on a certain level of user configuration to get the technology working. If such is the case, then the technology is not compact in its structure, and the components that make it up need configuration to work together. This puts more work on the part of the user, and hence devalues the technology.

- **Component Entry Points**: Of the many components that make up the technology, it is those components that are directly used by the application that are called entry components, and methods of these components are called entry methods. The less the number of such methods and components, the less coding is required on the developers’ part. Distributed object technologies attempt to hide these entry points through the use of proxy objects that intermediate communication. When calculating the entry points it is necessary to count these hidden entry points.
Given that we are evaluating ORB technologies using the same enterprise application, we will judge the ease of learning and use based on the number of entry points that our application uses, and in addition we will look at the effort that we spent trying to make our application distributed.

3.3.2.5 Portability and operating system issues

In previous sections of our thesis, we discussed the tools and operating systems that we need to build distributed object software. We have also mentioned that ORB technologies enable applications running on different platforms to communicate. In this section and as part of the criterion that we call portability, we try to identify the properties and conditions that help in evaluating a technology’s portability features. We will include the operating system issues as part of the concept of porting an application, and we will try to clarify some of the terms that we used throughout our thesis.

But first let us define what we mean by portability. Portability is the “overall process required to move – in part of as a whole – an application that is successfully operating in a certain environment to the target environment”. The porting is considered successful if the application retains its functionality after the process is completed”. This definition does not say anything about the portability of distributed applications, and this is expected, because a distributed application is an application that is operating in an environment, regardless of the size of the cluster. When porting such an application, the entire cluster – or a part of it – is moved to a different environment. It is expected that a successful porting of these components ensure the continued functionality of the entire cluster, and hence the application.

But there are certain inner points that need be clarified when dealing with the portability of distributed object software. In our experience with the ORB technologies that we subject to our thesis, we found a number of properties or issues that need be considered when porting such applications. We will list these properties and describe them, suggesting our understanding of what practices to follow when dealing with them:

- Network Protocol Compatibility: The underlying architecture of a distributed application is network communication. ORB technologies build their
communication protocols on top of existing protocols that are implemented into the operating system. It is important to check with the ORB specifications to find what networking protocols it uses. If these protocols do not exist on the target platform, the distributed application will obviously not work. Some protocols are implemented differently, and for that purpose a proper understanding of the operating system used should be achieved. All of the ORB technologies that we study here rely on TCP/IP to achieve their functionalities, and TCP/IP is compatibly implemented on all platforms that we know of. On the other hand, SOAP – the driving protocol for WebServices – has slightly different implementations across programming languages, but since we deal solely with the Java programming language, we face no problems with SOAP.

- **System-specific Differences**: when porting an application from an environment to another – and specifically from an operating system to another – we experience downgrades or upgrades in certain properties that the application inherits from that operating system. Such properties include performance, peripheral management – especially I/O management – in addition to some other properties. In other cases, the ORB implementation might be different on a different operating system, and since a distributed application has three logically independent layers, the lower two-layers – namely the operating system and the ORB layers – usually have different implementations in the different environments. We should be able to estimate the changes based on our knowledge of these platforms, and we advise to keep frequently communicating cluster nodes under the same platform.

It is most advisable to have the same platform and environment when working with a distributed application. Although mixing platforms and operating systems might be advisable in some situations – when an operating system is known to function better than others in a specific task – we recommend having a homogeneous cluster.

In a similar manner, we will evaluate the operating system and porting criterion in a comparative way, grading ORB technologies based on the number of implementations for the available operating systems, and looking at how they work interchangeably when
moved from one operating system to another. It is possible to take the entire application and run it on a different platform altogether, and recalculate all the empirical criteria mentioned above and compare the results across operating systems. If this is performed then the index of this criterion is the average of all these criteria indexes. We leave this for future work, and we consider only the qualitative differences in our current thesis.
4. Experimental Results

In this chapter we present our experimental results that we collected by running our enterprise application over a cluster of PC nodes.
4.1 Setup / Design

We will attempt in this section to give a brief overview of the resources we worked with.

It is obvious that working with distributed applications is more difficult than working with single-node applications, and it is the management of such an application in a distributed environment – a local area network in our case – that makes for most of this difficulty. A proper setup procedure is divided into sub-procedures, the details of which we state below, appending to every sub-procedure the situation we had and the ways we followed in our work.

We developed our application with the empirical criteria in mind. The application performs certain operations form within a framework, which in turn uses the ORB technology to achieve the services requested. Tracing the work of the application along the lines of these operations is the best way to evaluate the criteria under study. Logging these operations is achieved through writing to text files – or log files – for every criterion studied. The log files are separated in this manner to avoid concurrent logging, which is expected to happen since the application spawns many threads while working. As a general rule, every thread in the application at every node in the cluster should log to a separate file. The results for a given criterion are collected and merged – at the logical level – in hope of finding an acceptable evaluation for the criteria under study.

The compilation of the log files and their evaluation are performed in an analytical way through an automated analysis application. The results are reported in the form of criteria indexes, whose definition and procedure of computation are described in the respective sections of the “Evaluation Criteria” part of this thesis. In this section we list these index values for every technology in a tabular form. Every index is between 0 and 1, with higher index values representing better criteria evaluations. It should be noted that these values could change if experimentation takes place in different conditions, such as using different machines or different operating systems. We used 23 nodes to run the application, with one node as the master server. We distributed the remaining 22 nodes over 21 configurations of server and client nodes, starting with one server and finishing
with 21 servers in the cluster. We ran the application 10 times in every configuration for each ORB technology.

4.1.1 Operating System Installation

The operating system that we used for development purposed was Windows 2000 Professional, and we decided to host the master server of our enterprise application on a Windows 2000 Server machine.

4.1.2 Networkability

The machines in the network are configured for networking operations since the ORB technologies use TCP/IP to establish object communication. For the Windows and Linux operating systems networking comes pre-installed. It is recommended to have comparable communication times between the machines.

4.1.3 Software Installation

The software required for the technologies to work should be copied to the machines in the network. The effort to install and configure the software components grows linearly with the number of machines. In a mirrored networking environment standardizing the installation procedure greatly reduces setup time. In the following subsections we list the various software components that the ORB technologies require.

4.1.3.1 Database Server

MySQL is a free open-source database server that runs both on the Windows and the Linux operating systems. The installation of MySQL is straightforward

4.1.3.2 Apache Tomcat

Tomcat is the JSP servlet container required to deliver SOAP services, the key protocol in the WebServices ORB functionality. Tomcat is an open-source project. It is an all-Java server.
4.1.3.3 Java Runtime

JRE is a software component that comes with the Java Software Development Kit (JDK). It is required to run the application. It is a free software component, although not open-source.

4.1.3.4 NMake Utility

NMake allows writing configuration files to simplify the development process. It is not required when testing the application.

4.1.3.5 Microsoft SDK for Java

This is the SDK distributed by Microsoft to enable building applications that interface with COM on a Windows operating system. Installing this SDK on experimental machines is not required because all the tools required to run distributed applications come pre-installed under the Windows 2000 operating system.

4.1.4 Component Configuration

The various components that make up the software should be able to communicate with each other prior to running the application. The following configurations were required to make our application work.

4.1.4.1 Database Accessibility

The application is required to access the MySQL database instance installed on every machine in the network. The application accesses the local database instance, while loading the application to the network requires configuring the database server instances for remote login. A Java application should have access to a compatible JDBC driver. The JDBC driver for MySQL is freely available for download.
4.1.4.2 Tomcat Configuration

Initially Tomcat requires little configuration, since it automatically reserves port 8080 or any other port that the user selected during installation. Running the NT service of Tomcat at system startup is very useful in a big cluster, but it should be disabled when performing single-node development.

4.1.4.3 Application Configuration

The application configuration is stored in one script, which should be initialized with information about the network status, namely the number of machines and their addresses, and the address of the central server – the master server node – and some other database specific information. If the network daemon is to be used to launch the applications automatically then a special section of this script need be properly filled with daemon data. The configuration of the system is distributed, and a node first looks for configuration parameters in the local script and only if it cannot find them it goes to the master server script. Many parameters in the script can be made node-specific by appending to their names the underscore character – `_' – followed by the address of that node.
4.2 Results

In this section we will report the results, both experimental and qualitative.

4.2.1 Empirical Results

In this section we report only the quantitative results of running the enterprise application.

While the values may change, we do not expect the ratios to change. Table 2 gives the index values generated for the quantitative criteria, as explained in section 2.

<table>
<thead>
<tr>
<th>Criterion \ Technology</th>
<th>Java RMI</th>
<th>CORBA</th>
<th>WebServices</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance $\alpha_T$</td>
<td>0.6</td>
<td>0.7</td>
<td>0.4</td>
</tr>
<tr>
<td>Scalability $\beta_T$</td>
<td>0.4</td>
<td>0.8</td>
<td>0.7</td>
</tr>
<tr>
<td>Memory Management $\gamma_T$</td>
<td>0.7</td>
<td>0.8</td>
<td>0.6</td>
</tr>
<tr>
<td>Load balancing $\theta_T$</td>
<td>0.3</td>
<td>0.8</td>
<td>0.6</td>
</tr>
<tr>
<td>Fault-Tolerance $\psi_T$</td>
<td>0.5</td>
<td>0.8</td>
<td>0.6</td>
</tr>
<tr>
<td>Fail-Over $\omega_T$</td>
<td>0.4</td>
<td>0.7</td>
<td>0.5</td>
</tr>
<tr>
<td>Clustering $\xi_T$</td>
<td>0.8</td>
<td>0.9</td>
<td>0.75</td>
</tr>
</tbody>
</table>

The results are based on the processing and analysis of the log files that the application generated in a multiplicity of configurations.

RMI forwards its calls to the Java network library, and the latter forwards them to the libraries of the operating system – which happen to be written using the C language. CORBA forwards the ORB calls to the implementation, which could either be entirely outside the JVM or partially implemented using a different language. Therefore CORBA performs its network operations directly rather than through the Java library – if the implementation is outside Java. This is why CORBA shows superior performance in Table 2.
The distributed framework allows applications to scale to multiple cluster nodes. There is a limit set to this scaling since the JVM implementation on Windows 2000 is not very powerful and needs more resources. However, the reliability of CORBA proved central to the scalability index values. WebServices scaled as well but had their communication hindered by the nature of the JSP container that SOAP is implemented in. Apache Tomcat – or any other JSP container – places a layer on top of which the application is run, thereby servicing the clients through a layer that is not as thin as that of the RMI or CORBA naming servers. The RMI values resulted from the inability of the application to both scale out and handle large numbers of connections. Starving communication threads were noticed which limited scalability, and the RMI version suffered from partial communication blocks. The refresh rate assigned to the load-collection thread was extended by different ratios during the run. This can be traced back to the preemption algorithms of the Windows 2000 operating system. Different ratios are expected when the application is run on a different operating system. Therefore Web Services exhibit a higher scalability index in Table 2, followed by CORBA.

All three technologies delegate memory management to the JVM, but Web Services run on top of a JSP container and therefore have more Java code, while CORBA has the least Java code since it could have an implementation outside of the JVM. This is why Table 2 shows better memory management when using CORBA.

CORBA achieved an almost perfect balance. Sustainability is another criterion that can be studied future and is defined as the ability of an application to withstand incremental load that results in its crash. In order to achieve a good balance in the load distribution the application must survive an intense wave of client connections and processing requests. If the application crashes in the middle, the master node would delegate its role to the other cluster nodes. It was also noticed that when a node crashes, the other nodes soon fail due to the concentrated load that they now have to take. This is shown in Table 2 where CORBA scored higher load balancing values, while the results for RMI indicate a deficiency when handling load balancing.

4.2.2 Qualitative Results

Our qualitative evaluation is based on the experience of running the application and is recorded in the following observations.
4.2.2.1 Maintainability

We observed that DOC software that use the ORB technologies presented in this thesis and that have a layered application design are easily maintainable. The reason for this is that the calls into the ORB are separated from the business layers of the application, and therefore any change to the business or presentation layers can be performed regardless of the ORB technology used. In addition, using a framework similar to the one that we developed helps separate the common tasks and business operations from the ORB-specific ones. We observed that if a DOC application possesses this design, the time required to maintain business operations within the application is comparable to that of a single-node application.

In addition, we observed that the layers of the DOC application that use the ORB technologies are best maintained if the ORB technology is implemented as separate components that are external to the programming language. For this reason, using Java RMI requires more maintenance than CORBA and Web Services, since Java RMI is implemented as components within the programming language libraries. CORBA is implemented as independent components that are loaded at run-time, which makes the maintenance of the DOC application easier.

4.2.2.2 Testability

When testing a DOC application that uses an ORB technology, component testing can be best achieved when the DOC application uses a design similar to that of our application. Separating the business logic from the calls into the ORB makes it easier to test the application. However, testing the application components that use the ORB technology is easiest to perform when the ORB technology is implemented within the libraries of the programming language. Therefore testing DOC components that use Java RMI is easier than the other ORB technologies since Java RMI is implemented within the Java programming language. Components that use CORBA are the least testable because CORBA is implemented as external components written in a different language.
4.2.2.3 Repeatability and Reproducibility

We observed that the layers of DOC software that perform calls into the ORB libraries can be repeated and reproduced in different DOC applications when the ORB is fully Object Oriented and when it integrates directly with the programming language used. We observed that the layers that use Java RMI could be reproduced with minimal effort since Java RMI is tightly coupled with the Java programming language, while the same layers that use CORBA require more code. The reason for this is the loose coupling between CORBA and the programming languages that it is used from. We observed that components using Web Services require the most programming effort to reproduce, since Web Services are implemented within other network protocols such as HTTP and the Simple Mail Transfer Protocol (SMTP).

4.2.2.4 Ease of Learning

When developing DOC software, we observed that using Web Services in the application requires familiarity with network protocols such as HTTP, especially when debugging the application. We also observed that CORBA depends on IDL to interface the application components. The Java compiler can generate IDL from Java application components, and therefore using CORBA from DOC software is easier than using Web Services. Java RMI is the easiest of the ORB technologies to learn, since it is tightly coupled with the programming language.

4.2.2.5 Portability

Building DOC software using the Java programming language makes the application code portable across heterogeneous operating systems and platforms. We observed that DOC components that make calls into the ORB technology libraries reduce the DOC application's portability. The reason for this is the degree of coupling of the ORB libraries with the underlying operating system. When using Java RMI and CORBA, the choice of the port used by the naming server depends on the privileges assigned to the application by the operating system. Web Services use the same port that the JSP container uses, which in turn is assigned by the system administrator. We also observed
that the CORBA naming server that is used by DOC software to lookup object references is implemented in a different programming language and is external to the Java Virtual Machine (JVM) within which the application is running. The naming server is an application that maps registered object names to object locations across the network. The naming server that Web Services use is internal to the JSP container, and therefore the portability of Web Services depends on the portability of the JSP container. We observed that Java RMI is the most portable among the ORB technologies studied, while CORBA is the least portable.
4.3 Discussion

This section will be divided into a general evaluation part, in which we look at the criteria from a global perspective. The second part will cover the evaluations that we reported for the empirical criteria, followed by a third part for the qualitative criteria evaluation. In the final part we will discuss the effect of these criteria on each other, and suggest procedures of work that would help clear the link between these criteria.

4.3.1 General Evaluation

When performing an evaluative approach to object technologies, particularly distributed technologies, we are always faced with one factors that hinder the correctness of our expectations. These factors are there whenever objects are used to solve problems similar to the ones encountered in the computer science domain. Distributed object software adds its own factors that add up to the total complexity that is associated with evaluating object technologies. We will look at these factors first, and then discuss what we feel is a general approach to evaluating the criteria of distributed object software.

- **Resource Limitations:** always a factor in any software study, this factor affects most distributed software evaluation. Resources are considered to be the totality of components – both soft and hard – that are required to make software technologies functional, regardless of the quantities or qualities that they are available in. For example, disk storage is such a resource, and it affects the availability of swap space for memory operations, the size by which the database can grow, the space allocated for log files, and so forth. Another example is memory, the buffers that are used to hold the data-structures of the technology, and how fast these can be accessed by the application’s framework. There are many resources in a distributed environment that are not required or included in single-node environments, but these same resources are used by both object technologies and non-object oriented network technologies. The need for more of these resources on the side of object software is explained by the mere structure of the object-oriented model. The way objects are handled inside the execution framework, and the process of marshalling these objects – either as parameters or
as instance copies – implies that such resources are needed and should be of such quality and availability. Unlike some cases of single-node technologies, the absence of these resources make it not only difficult to provide distributed services, but rather impossible.

- **Troubleshooting Problems:** This is a major factor affecting our decisions concerning distributed object technologies’ criteria. It would sound very intuitive and straightforward to build distributed applications with any of the technologies we study in this thesis. As a matter of fact the documentation sections of these technologies each states that it is seamless to do this using the technology provided. However we have noticed that the examples and techniques provided by this documentation include only a “one server one client” model, and no other functionality is explained but that of bringing some parameter from one machine to another and sending it back in a modified form. The many troubles that are inherently associated with deploying and running distributed object software are not included as part of such documentations, and very few – if any – literature studies and papers discuss these difficulties and provide solutions for them. We faced many problems during our work, and the best way to understand these problems and our associated solutions is to re-live such an experience at a similar scale. We were able to troubleshoot some of these problems, and these were the ones that hindered our resulting evaluations in the first place. Many others still remain, and we believe that some of those remaining may not be solvable.

- **Analysis-Thoroughness Tradeoff:** In every study that one makes, there has to be a tradeoff between the length of time for which analysis is to be performed and the degree of completeness that analysis ends up having. We noticed this factor and were faced with this same problem. In such situations, and indeed in many others, an iterative approach to the problem at hand seems the best practice. Solutions are never complete, at least in such fields as the one we intended to study. There is always a way through which a distributed component can perform better, or possibly fine-tuned to reduce network load and come up with less memory-consuming data structures. The hunt for completeness never ends, and this is true even with non-distributed software. The only thing that we advise is to follow an
iterative approach that would at first generate inaccurate results but in a very fast way, and this tradeoff between time and accuracy can then be worked for as long as the project or research can go. The final – or conclusive – results would definitely hold a complete picture that, although not as accurate as desired, has the entire aspects of the study included.

4.3.2 Intra-Criteria Effects

The criteria that we evaluated in this thesis are not completely independent in every aspect. While they can be separated and studied individually and still get similar results to the ones we report in this thesis, we must explain that these criteria affect each other in some ways. To better understand these ways, we have decided to devote a section of our results to explaining them. We will report for every criterion the other criteria that it affects and how it affects them. While this is not backed up with conclusive data or analysis, we feel that such a dependency not only exists but also affects the way these criteria are measured.

4.3.2.1 Speed and Performance

We were able to identify relationships between this criterion and the following criteria:

- *Memory Management*: this is true because the faster the application runs the more memory it requires. Put differently, the more memory a technology uses to provide the services requested, the faster it should operate and the more performance it will have, provided that this memory is used to host the data structures that this technology uses. We know that algorithms attempt to load up their data structures into more memory in order to reduce the time needed to process them.

- *Load Balancing*: surely this is true. There is a strong relationship between performance and load balancing, since the latter reduces node-based operations, or in other words it balances such operations over all cluster server-nodes. Hence
the better load balancing is done, the better the application can leverage the powers of the distributed object technologies that it is using.

### 4.3.2.2 Scalability

Scalability is very much dependent on the following criteria and can itself affect them in many ways:

- **Clustering:** the better a distributed environment is clustered and its nodes’ cooperation promoted, the more scalable the application becomes, since scalability is the ability to grow, and an environment cannot grow if the size is not properly managed.
- **Maintainability:** The more scalable the application is, the easier it becomes to maintain it. The technologies that the application relies on to do its work become easily maintainable because growing the clusters.

### 4.3.2.3 Fault-Tolerance

Fault-tolerance depends on many factors that are also present in some other criteria that we list below:

- **Fail Over:** when the distributed object technology fails-over, the time needed to retain functionality might be noticed by other nodes, and these nodes require fault-tolerant algorithms that would keep the processes running until the new standby server takes over.
- **Testability:** the more fault-tolerant the application is, the harder it is to test it against faults through fault-injection. If testability of the application is more important that its fault-tolerance, then the technology-dependent parts of the application should be isolated and tested away from fault-prone environments.
5. Conclusion and Future Work

In this thesis we compared three ORB technologies that are most used in developing DOC software. CORBA, Java RMI, and Web Services were used within DOC application and their properties were evaluated for various criteria such as performance, load balancing and scalability. We also recorded our observations for certain qualitative criteria such as testability and portability. Our quantitative results and our qualitative observations show that no single ORB technology can be used alone in DOC software development. For example, CORBA is superior when load balancing is used, while Java RMI is the best technology to use when rapid deployment and development are required, while WebServices are the best solution for scalability and portability. The choice of ORB technology depends on the requirements of the DOC software. Our results and observations make it possible to choose the ORB technology at the design stages, and this reduces the time required to build DOC software.

There is a lot of work to be done if an extension of this thesis is desired. We find many aspects of our thesis incomplete, and we would like to see a comparison between the many implementations of CORBA. DCOM should be considered as well, and while we failed to put DCOM at the center of our thesis, we know that it is possible to implement these criteria in the DCOM model. Another thing that we would like to see is the inclusion of the new .Net framework in a similar survey. The .Net programming model introduces a new distribution technology that seems to behave similar to RMI. It would be interesting to look at this new technology, especially because it – like WebServices – runs on top of another framework; the .Net framework.

Another thing that can be studied is the introduction of a new socket-based ORB technology that would be custom-built to leverage the power of network protocols other than TCP/IP. We would expect UPD to play a competitive role, and an interesting study could be conducted to look at the various network protocols using this same custom technology. Such a work would defiantly bring new insight into distributed software development, and would move the center of gravity away from programming technology clients to that of building frameworks and protocols. We can never understand a technology unless we create a similar one!
References


[JuZi00] M.B. Juric, A. Zivkovic, I. Rozman, Comparison of CORBA and Java RMI Based on Performance Analysis, MHSS'98, 1998


[Zah00] R. Zahavi, Enterprise Application Integration with CORBA Component and Web-Based Solutions, John Wiley & Sons, 2000