A new message-based protocol for building a platform and language independent distributed object model

Hazem El Ashmawi

Follow this and additional works at: https://fount.aucegypt.edu/retro_etds

Recommended Citation

APA Citation

MLA Citation

This Thesis is brought to you for free and open access by AUC Knowledge Fountain. It has been accepted for inclusion in Archived Theses and Dissertations by an authorized administrator of AUC Knowledge Fountain. For more information, please contact mark.muehlhaeusler@aucegypt.edu.
A NEW MESSAGE-BASED PROTOCOL FOR BUILDING A PLATFORM AND LANGUAGE INDEPENDENT DISTRIBUTED OBJECT MODEL

A Thesis Submitted to

Computer Science Department
in partial fulfillment of the requirements for
the degree of Master of Science

by

Hazem S. El Ashmawi
under the supervision of

Dr. Ahmed Sameh

December/2002
The American University in Cairo

A NEW MESSAGE-BASED PROTOCOL FOR BUILDING A
PLATFORM AND LANGUAGE INDEPENDENT DISTRIBUTED
OBJECT MODEL

A Thesis Submitted by Hazem S. El Ashmawi
to Department of Computer Science
December/2002

in partial fulfillment of the requirements for
The degree of Master of Science

has been approved by

Dr. Ahmed Sameh
Thesis Committee Chair / Adviser ________________________________
Affiliation ____________________________________________________

Dr. Awad Khalil
Thesis Reader / Examiner ________________________________
Affiliation ________________________________________________

Dr. Amr El Kadi
Thesis Reader / Examiner ________________________________
Affiliation ________________________________________________

Dr. Abdel Badie Salem
Thesis Reader / Examiner ________________________________
Affiliation ________________________________________________

_________________________ _____________________________
Department Chair / Date Dean
Program Director
Acknowledgement

I would like to express my deep gratitude to Professor Dr. Ahmed Sameh for his continuous and invaluable guidance all through the making of this work and also for his patience and utmost support.

I would also like to thank Professor Dr. Awad Khalil and Professor Dr. Amr El Kadi for their precious advices and tips during the proposal and through the course of the thesis.

Last but not least, I would like to thank all my family for supporting me and being most patient and loving throughout the whole period of my studies.
ABSTRACT

Standardization of messaging topologies for communication in recent distributed object computing architectures is becoming more and more inevitable. The emergence of a structured and flexible document model as XML has made an entry point towards this goal. In this thesis, we are utilizing the flexibility of XML and the simplicity of low-level socket communication to build a generalized messaging model that provides a basis for standardization and supports interoperability among existing distributed object computing architectures. The proposed system is composed of the basic components of a distributed architecture constituting a number of broker components acting as naming services and client/server objects. All components share the same features of having built-in support for XML parsing and communicating with sockets. The proposed model is language independent, so we used heterogeneous programming languages to model various components and test its feasibility. The measurement of invocation time is used for testing to provide an overview of the performance and overhead incurred by the system. Different runs with different types of components using direct and broadcast addressing are tested on multi-node setups and invocation times are measured as round trips from the client’s request to the server’s response.
# Table Of Contents

1. **Introduction And Background**
   1.1 Definition of a Distributed System
   1.2 Benefits of a Distributed Model
   1.3 Design Challenges of Distributed Systems
   1.4 Anatomy of a Distributed Application
   1.5 Prerequisites for Developing Distributed Applications
   1.6 The Problem
   1.7 Motivation
   1.8 Contribution
   1.9 The Outline of the Thesis

2. **Historical overview of distributed systems**
   2.1 Evolution of the concept
   2.2 Evolution of IPC models in distributed systems
   2.3 Issues of concern in modern distributed environments
   2.4 Summary

3. **Overview Of Some Modern Distributed Object Computing Models**
   3.1 Interoperation Layering Model
   3.2 Microsoft Distributed Component Object Model (COM+)
   3.3 Common Object Request Broker (CORBA)
   3.4 Java Remote Method Invocation (RMI)
   3.5 Enterprise Java Beans (EJB)
   3.6 Microsoft Simple Object Access Protocol (SOAP)
   3.7 Summary

4. **The Proposed Model**
   4.1 Introduction
4.2 Platform Preference

4.3 Programming Language

4.4 Complexity

4.5 Security

4.6 Serialization

4.7 Compression

4.8 Versioning

4.9 Flexibility

4.10 Interoperability and Standardization

4.11 Components

4.12 Communication Channels

4.13 Features of The Model

4.14 Summary

5. System Design

5.1 Basic Setup

5.2 Communication Scenarios

5.3 The RPC Protocol Formats

5.4 Summary

6. Implementation And Benchmarking

6.1 Introduction

6.2 Methodology

6.3 Development Tools

6.4 System Components

6.5 The Java RMI System

6.6 Problem Setup

6.7 Summary

7. Results

7.1 Introduction

7.2 Observations
List of Tables

Table (1)  XML and Component Integration Technologies_________24
Table (2)  Comparison Between Distributed Object Technologies__52
Table (3)  Instantiation and null RPC times on the same node_______93
Table (4)  Instantiation and null RPC times on remote nodes________93
## List of Figures

<p>| Figure (1) | Mainframe Architecture | 8  |
| Figure (2) | Client/Server Architecture | 9  |
| Figure (3) | Multitier Client/Server Architecture | 10 |
| Figure (4) | Distributed Systems Architecture | 11 |
| Figure (5) | Sun RPC Message Format: XDR Specification | 13 |
| Figure (6) | DCOM Communication scenario | 27 |
| Figure (7) | CORBA Communication Scenario | 35 |
| Figure (8) | The J2EE Object Model | 41 |
| Figure (9) | DCOM Authorization scenario | 57 |
| Figure (10) | Overall System Design Diagram | 71 |
| Figure (11) | Transparent Component Addressing | 76 |
| Figure (12) | Targeted Component Addressing | 76 |
| Figure (13) | Direct Component Addressing | 77 |
| Figure (14) | Parallel Processing | 77 |
| Figure (15) | The Master Component Interface | 89 |
| Figure (16) | Instantiation times using Direct Addressing | 94 |
| Figure (17) | Instantiation times using Broadcast Addressing | 95 |
| Figure (18) | Instantiation times for local components | 96 |
| Figure (19) | Instantiation times for remote components | 96 |
| Figure (20) | null RPC times | 97 |</p>
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACL</td>
<td>Access Control List</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>ASCII</td>
<td>American Standard Code for Information Interchange</td>
</tr>
<tr>
<td>CDR</td>
<td>Common Data Representation</td>
</tr>
<tr>
<td>CLSID</td>
<td>Class Identifier</td>
</tr>
<tr>
<td>COM</td>
<td>Component Object Model</td>
</tr>
<tr>
<td>CORBA</td>
<td>Common Object Request Broker Architecture</td>
</tr>
<tr>
<td>CSS</td>
<td>Cascading Style Sheets</td>
</tr>
<tr>
<td>DCE</td>
<td>Distributed Computing Environment</td>
</tr>
<tr>
<td>DCOM</td>
<td>Distributed Component Object Model</td>
</tr>
<tr>
<td>DII</td>
<td>Dynamic Invocation Interface</td>
</tr>
<tr>
<td>DOM</td>
<td>Document Object Model</td>
</tr>
<tr>
<td>DOS</td>
<td>Disk Operating System</td>
</tr>
<tr>
<td>DTD</td>
<td>Document Type Definition</td>
</tr>
<tr>
<td>EJB</td>
<td>Enterprise Java Beans</td>
</tr>
<tr>
<td>HTML</td>
<td>Hypertext Markup Language</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext Transfer Protocol</td>
</tr>
<tr>
<td>IDL</td>
<td>Interface Definition Language</td>
</tr>
<tr>
<td>IIOP</td>
<td>Internet InterORB Protocol</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
</tr>
<tr>
<td>IPC</td>
<td>Inter-process Communication</td>
</tr>
<tr>
<td>J2EE</td>
<td>Java 2 Enterprise Edition</td>
</tr>
<tr>
<td>JDK</td>
<td>Java Developer Kit</td>
</tr>
<tr>
<td>JRMP</td>
<td>Java Remote Messaging Protocol</td>
</tr>
<tr>
<td>JSP</td>
<td>Java Server Pages</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Full Form</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td>LAN</td>
<td>Local Area Network</td>
</tr>
<tr>
<td>LPC</td>
<td>Local Procedure Call</td>
</tr>
<tr>
<td>MFC</td>
<td>Microsoft Foundation Classes</td>
</tr>
<tr>
<td>MSDN</td>
<td>Microsoft Developer Network</td>
</tr>
<tr>
<td>MSMQ</td>
<td>Microsoft Message Queue</td>
</tr>
<tr>
<td>MSXML</td>
<td>Microsoft XML parser</td>
</tr>
<tr>
<td>MTS</td>
<td>Microsoft Transaction Server</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>ORB</td>
<td>Object Request Broker</td>
</tr>
<tr>
<td>OSF</td>
<td>Open Software Foundation</td>
</tr>
<tr>
<td>PHP</td>
<td>Personal Homepage Tools</td>
</tr>
<tr>
<td>RMI</td>
<td>Remote Method Invocation</td>
</tr>
<tr>
<td>RPC</td>
<td>Remote Procedure Call</td>
</tr>
<tr>
<td>SAX</td>
<td>Simple API for XML</td>
</tr>
<tr>
<td>SGML</td>
<td>Standard Generalized Markup Language</td>
</tr>
<tr>
<td>SOAP</td>
<td>Simple Object Access Protocol</td>
</tr>
<tr>
<td>TCP</td>
<td>Transfer Control Protocol</td>
</tr>
<tr>
<td>UDP</td>
<td>User Datagram Protocol</td>
</tr>
<tr>
<td>VB</td>
<td>Visual Basic</td>
</tr>
<tr>
<td>VC</td>
<td>Visual C++</td>
</tr>
<tr>
<td>W3C</td>
<td>World Wide Web Consortium</td>
</tr>
<tr>
<td>WAN</td>
<td>Wide Area Network</td>
</tr>
<tr>
<td>XDR</td>
<td>Extended Data Representation</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
<tr>
<td>XSL</td>
<td>Extensible Style Sheet Language</td>
</tr>
<tr>
<td>XSLT</td>
<td>Extensible Style Sheet Language for Transforms</td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION AND BACKGROUND

1.1 Definition of a Distributed System:

Tannenbaum and Van Renesse [30] define a Distributed System as one that looks to its users like an ordinary centralized system but runs on multiple independent CPUs. The key concept of which is transparency. The use of multiple processors should be transparent to the user and consequently the user should interpret the system as a uniprocessor one.

1.2 Benefits of a Distributed Model:

1.2.1 Resource Sharing:

One of the most important issues in today’s computing practice is to be able to share different kinds of resources with other users depending on the complexity and size of the problem at hand. For instance, at some time a user might want to use more processing power than available or to use more disk space or to access a remote centralized data source. This can only be made available through a tightly integrated distributed model which allows different users of the system to be able to utilize the system’s resources as required.

1.2.2. Fault Tolerance:

Distributed systems allow for replicating different services and can perform similar tasks redundantly which allows for protecting system users from unexpected hardware and/or software failures. It also results in an increased overall system availability and better response time.

1.2.3 Cooperation:

Distributed Systems allow users to work more collaboratively and to perform certain tasks in conjunction with each other by combining
concepts such as wide area networking, multiple workstations and the presence of graphical user interfaces. Applications that demonstrate these concepts include email systems, conferencing systems and web databases.

1.2.4 Parallelism:

Due to the increase in processor abilities and processing power, combining several processors to solve some problem would yield some speedup. The concept of distribution allows for this by combining the resources available from several components in the system to perform a certain task. Before introducing Distributed Systems, parallel processing was only available to extremely expensive supercomputers harboring a processor pool.

1.3 Design Challenges of Distributed Systems:

1.3.1 Software Complexity:

The development and implementation of a distributed application which provides support for fault tolerance and transparency is inherently complex and has to handle every aspect of system drawbacks and faults. Moreover, debugging error tracing in such a system provides a challenging problem at hand.

1.3.2 Network Saturation:

A Distributed System with increasing number of nodes can exert a severe network and communication overhead especially in communication intense conditions. Under such conditions, the system may show unacceptable performance even to users not involved in the condition causing the bottleneck. Even though, with the increasing bandwidths of the existing networking technologies, network overloads may occur according to inter-nodal communications densities and network resources demand.
1.3.3 Failures:
In Distributed Systems, failure of system components or nodes is not an uncommon condition. It may be caused by a hardware failure, a software error or faulty user intervention. Thus, a well-designed Distributed System should be able to handle such conditions to the maximum possible else a single component failure may bring the whole system to a stop. Several ways are present to deal with such conditions such as replication of components and load balancing techniques.

1.3.4 Inconsistencies:
With the presence of replication of certain components of a Distributed System and the concurrent connections to such replicable components, arises the problem of inconsistency. The state of such a replicable component may differ from one replica to another keeping the system in a state of inconsistency. Schneider [28] stated that dealing with such a problem should provide a way for such replicated components to make regular updates between each other or to fire state change events to notify other replicas of the changes that occurred.

1.4 Anatomy of a Distributed Application:
A distributed application is built upon several layers that may vary depending on the complexity of the application model and the underlying Operating System. The network layer at the lowest level, network protocols like TCP/IP, higher-level services such as directory services and security services and finally the distributed application components run on top of these layers [11]. A distributed application can be divided into the following parts:

1.4.1 Processes:
A process is a sequence of steps written in any programming language to be executed in the operating system. A process has access to
the resources of the workstations it is running on. It can also serve one application or many applications.

1.4.2 Threads:

Every running process has at least one thread of control. Some processes might have multiple threads of control running independently of each other if the operating system permits. One thread might have a socket listening for input or monitor changes in the file system and the other thread acts by performing some tasks according to data received or changes recorded. Most of the time, threads running in the same process will require some synchronization.

1.4.3 Objects:

An object is a group of related data with methods available to handle this data. A process can be made up of one or more objects and these objects can be accessed by one or more threads within the process. With the introduction of distributed object technology objects might be spread among multiple nodes.

1.4.4 Agents:

An agent is a higher-level system component defined around a particular function or utility in the system [12]. An agent is usually composed of one or more related objects and can be distributed among multiple processes. The distributed system can be thought of as multiple functioning agents cooperating together to perform a certain task.

1.5 Prerequisites for Developing Distributed Applications:

1.5.1 Data Distribution and Partitioning Functions:

Computational tasks can be distributed based on the data needs of the application. In computing intense applications, one might prefer to
partition the system according to computing functionality into multiple logical units, while in a data intense application partitioning of data in such a way to minimize data transfers over the network and maximize data availability for each unit. In partitioning, the most important factors to be taken into consideration are the network throughput in comparison to local processing time needed to accomplish a task.

1.5.2 Flexible, Extensible Communication Protocols:

Agents comprising the distributed system must have a flexible way to perform communication with each other. They might also be forced to use non-standard communication protocols as dictated by some legacy systems, so they must have a way to incorporate new protocols into the system or have some way of attaining new communication possibilities in the system.

1.5.3 Multithreading:

Multithreading capabilities in the operating system is a must in distributed environments. Agents almost always need threading to perform some tasks while blocking on I/O or to handle requests from multiple different agents at once. Moreover, in heterogeneous environments with nodes having different processing capabilities, multithreading would serve to achieve better load balancing and resource usage.

1.5.4 Security:

One of the most important issues in distributed environments is securing access to system agents. The minimum required security levels would be authenticating the source of an agent, defining access levels for an agent to functionalities of other agents and to secure data transmission
Introduction and Background

across the network where certain messages might carry sensitive data like credit card information or bank account transactions.

1.6 The Problem:

The thesis addresses the problems of complexity, flexibility, standardization and interoperability that show with modern distributed object computing models: The goal of this thesis is to develop simple mechanisms for standardizing inter-process communication and messaging without violating cross boundary interoperability between peers in a decentralized, distributed environment using XML. The work verifies that XML is the best transfer medium for inter-component messaging protocol and also provides a rich database for system related information.

Providing TCP and UDP sockets as the communication channels allows for the flexibility and extensibility of the system and also avoids the overhead incurred by designing a proprietary transfer protocol. Its simplicity also meets the requirements for sending simple ASCII XML messages.

1.7 Motivation:

Existing systems have problems and the motivation of this thesis is to propose a solution for different types of problems by using XML over socket communication. These problems include complexity, flexibility, platform and language independence, interoperability, serialization and standardization.

1.8 Objectives and Goals:

The objective of this work is to present a new foundation for distributed object modeling based on XML message communication and
sockets to achieve higher levels of simplicity, flexibility, interoperability, and standardization in modern distributed object architectures.

1.9 Contribution:

This thesis demonstrates that XML and a mix of TCP and UDP sockets can provide complete basis for the development of a mature distributed object model. This approach to modern distributed computing field is adapted to provide all essential aspects that constitute a fully functioning distributed model and provides solutions and workarounds for some of the most annoying design challenges in these types of systems.

1.10 The Outline of the Thesis:

Chapter 1 introduces the concept of distributed systems and outlines the components and the inherent problems involved in its development. Chapter 2 provides an outline of the different communication mechanisms and distributed object architectures. Chapter 3 gives an overview of the most well known distributed object architectures and their pros and cons. Chapter 4 presents the proposed system that uses XML and socket communication and outlines the benefits gained from building such a system. Chapter 5 provides the details and requirements for the design of the proposed system. Chapter 6 describes the implementation, programming requirements, measurements used and setup for testing the proposed system. Chapter 7 presents the results obtained from test measurements and provides an analysis for these results. Chapter 8 outlines the conclusion, open issues and future work to be done towards more maturation of such a system. The appendix introduces XML and outlines different technologies used with such a language.
CHAPTER 2

HISTORICAL OVERVIEW OF DISTRIBUTED SYSTEMS

2.1 Evolution of the Concept:

Since early stages of Distributed Systems research, computer scientists have passed through several eras of development and enhancement based on the increasing needs and demands of computer users in this evolving market.

2.1.1 Main Frame (Monolithic) Systems:

At the beginning, there was the monolithic mainframe system which had all the processing (database, business rules and interface) on one machine accessed by dummy terminals [11]. There was no real distribution as in today’s sense (Figure 1).

![Figure (1) Mainframe Architecture](image-url)
2.1.2 Client/Server Systems:

Then came the revolutionary client/server architecture that separated the database layer from the interface (client side) layer. This had the advantage of isolating the storage and data maintenance from the data entry and retrieval side [10]. Each of the 2 components was separate and an upgrade in one would not need a redistribution of the other (Figure 2).

![Client/Server architecture diagram]

*Figure (2) Client/Server architecture*

2.1.3 Multi-tier Client/Server Systems:

The emergence of client server lead to the evolution of multi-tier client/server systems in which there was further separation and isolation of the business rules layer to be on a third node which made a leap forward in load distribution and balancing (Figure 3) [11].
With further progress in the field of multitier systems, the emergence of completely distributed systems was inevitable. These systems had completely distributed components with even backup components and security systems (Figure 4).

Ever since, there have been large enhancements in the field of distributed systems and environments contributed to by several of the most prominent organizations in the business field [10]. Following is a review of multiple architectures and implementations of IPC mechanisms and object communication protocols that were used to verify the concept of distribution and the enhancements done towards standardization.
2.2 Evolution of IPC Models in Distributed Systems:

2.2.1 Socket Communication:

The very basic and straightforward way to make applications communicate with each other was the use of sockets. A socket is created by an application which binds a particular address to it called a port and listens to incoming messages. There are several different types of sockets each with its own address type. The most commonly used types of sockets for communicating applications on two machines are the Transfer Control Protocol (TCP) sockets form the foundation of internet communications and web servers nowadays. User Datagram (UDP) sockets are another type of commonly used sockets, also called the Unreliable Datagram Sockets. UDP sockets are commonly used for broadcast systems, video and sound streaming and are mostly suited for short packet message systems [20] The most obvious disadvantage of
Historical Overview Of Distributed Systems

using sockets is the overhead of low level coding used to setup the communication and the need to deal with low level error handling. On the other hand socket programming provides the advantage of language and platform independence and the flexibility to build the most suitable implementation without the runtime overhead of other schemes.

2.2.2 Remote Procedure Calls (RPC):

The Open Network Remote Procedure Call standard or ONC-RPC for short was developed based on the remote procedure call developed by Sun Microsystems since early 1980s [20]. It has become the standard among UNIX systems. RPC makes the paradigm of network programming look like ordinary procedure calls. The client makes a request of the server and the server responds with the result; the caller calls the procedure call passing in necessary arguments and the call returns the result when finished. The remote procedure call interface is specified using a special language called remote procedure description language. A protocol compiler converts these language files into source modules that the programmer links with. These modules together with the C runtime library perform the necessary network functions. RPC handles finding the remote server by the aid of a well-known server that acts as a directory of services running on the machine. This server is called the portmap server. ONC-RPC supports both TCP and UDP sockets and also stream based procedure calls. ONC-RPC uses a special data format for marshaling and de-marshaling arguments from a client to a server called the Extended Data Representation XDR. The XDR representation of data is a binary representation that carries several pieces of information about the message in a binary format and in a specific order. The most important of this information is the call type, version of RPC, remote program number, version of remote program, remote procedure number
and authorization information. In this context, versioning using ONC-RPC was a valid issue and also security was taken care of, in a way, in this implementation (Figure 5).

**Sun RPC Message Format: XDR Specification**

```c
enum msg_type { /* RPC message type constants */
    CALL = 0;
    REPLY = 1;
};

struct rpc_msg { /* format of a RPC message */
    unsigned int mesg_id; /* used to match reply to call */
    union switch {msg_type msgt| {
        case CALL: call_body cbody;
        case REPLY: reply_body rbody;
    } body;
};

struct call_body { /* format of RPC CALL */
    u_int rpcVers; /* which version of RPC? */
    u_int rprog; /* remote program number */
    u_int rprogVers; /* version number of remote prog */
    u_int rproc; /* number of remote procedure */
    opaque_auth cred; /* credentials for called auth. */
    opaque_auth verif; /* authentication verifier */
    /* ARG5 */
};
```

*Figure (5) Sun RPC Message Format: XDR Specification [1]*

### 2.2.3 Open Soft Foundation (OSF) Distributed Computing Environment (DCE):

DCE-RPC is a slightly different variant of remote procedure calls, developed by the open software foundation (OSF) now called the Open Group, the developers of Motif [22]. DCE-RPC was built to take the RPC standard to a higher level of transparency with object support. This standard was the base of Microsoft’s distribution scheme for DCOM. However, DCE-RPC is neither compatible with RPC, nor compatible with Microsoft’s version of DCE-RPC. Moreover, RPC and DCE-RPC are out of support as their sponsors have moved on to supporting CORBA development.
2.2.4 Common Object Request Broker Architecture (CORBA):

CORBA is a standardized scheme for constructing distributed object based applications, allowing components to communicate with one another regardless of platform or implementation language developed by the Object Management Group (OMG) since 1991 when the CORBA 1.1 specification was released [22]. The OMG has agreed on standard ways of mapping IDL constructs onto several computer languages, including Ada, C, C++, COBOL, Common Lisp, Java and Smalltalk. Other organizations have established less formalized mappings for working with other languages such as Perl, Python and TCL. CORBA most strongly resembles RPC allowing clients to request remote function calls. In CORBA locations of servers are managed by an object reference which masks the host/socket information provided by RPCs. CORBA has a framework for adding additional services. CORBA provides a dynamic invocation interface (DII) which provides a way for discovering what methods an object exposes at runtime rather than compiling an IDL interface at design time. CORBA supports multiple network protocols and shared memory in addition. More details on CORBA will be presented later.

2.2.5 Microsoft Distributed Component Object Model (DCOM):

This is Microsoft’s system for distributed computing similar to CORBA in architecture but with some differences. Microsoft began working on distributed COM objects in mid 1990s with DCOM and eventually the union of the technology with Microsoft Transaction Server transactional capabilities resulted into the new COM+ technology. Like CORBA, DCOM provides a relatively language independent IDL, however, Microsoft IDL commonly includes quite detailed system configuration information, indicating data that would normally be
managed using the Windows Registry. As the product of a single vendor system, COM and DCOM are provided with the benefit of Win32 facilities like the Windows Registry and the uniform availability of identical implementations of dispatchers and services [23]. If the need exists to combine CORBA and DCOM objects in the same system, there are bridge products that allow communication back and forth and OMG is actively working on such interoperability. More details on DCOM/COM+ will be presented later.

2.2.6 JAVA Remote Method Invocation (RMI):

This is Sun’s Java environment support for distributed object systems; it does not reach the sophistication and complexity of CORBA services. Since it only needs to support Java applications, it is simpler to use than CORBA. CORBA and RMI share the property of being relatively platform-independent, but Java RMI only works with Java programs, i.e. language dependent. Recent efforts have involved implementing RMI with the IIOP, the same network protocol used with CORBA for ORB to ORB communications. This may represent something of convergence of the two technologies. More details on Java RMI will be presented later.

2.2.7 Enterprise Java Beans (EJB):

Enterprise Java Beans (EJB) provide a more or less similar scheme to CORBA and DCOM, with a standardized and fairly sophisticated way of invoking distributed components that includes a framework for handling persistent data. EJB are restricted to the Java language as is the case with Java RMI and suffer from all performance problems to which Java based systems are susceptible. However, as Java compilers improve, performance should become less of a problem. There are a number of EJB implementations which make it possible that EJB may prove useful.
Also, the fact that EJB servers use RMI-IIOP for communication make it possible that CORBA clients can communicate with EJB server objects. More details on EJB will be presented later.

2.2.8 Microsoft Simple Object Access Protocol (SOAP):

Microsoft has introduced the Simple Object Access Protocol (SOAP) and together with several developers from multiple companies a standard for SOAP is being developed at the W3 consortium. The essence of SOAP is that RPC are requested using the HTTP protocol with data passed in XML form. SOAP differs from CORBA in that the messages that are transmitted are either represented as text or in a form trivially transformed into text, which means that the data being passed around may be easily examined and transformed while in transit. This can be advantageous in debugging the system, but is also a large security hole. More details on SOAP will be presented later.

2.3 ISSUES OF CONCERN IN MODERN DISTRIBUTED ENVIRONMENTS

2.3.1 Language preference:

The programming language represents the first obstacle in choosing the architecture that most suites the developer’s needs. It is wise to say that whenever one is stuck to JAVA and do not plan by any means to expose classes or interfaces to other languages, one can safely use the JAVA RMI which will then be more light weight and fast enough without the unneeded overhead of either CORBA or DCOM runtime.

If a developer prefers to develop in C++, then he can choose between CORBA and DCOM according to platform issues, while if he plans to use COM oriented rapid application development environment (RAD) then there is no dispense with DCOM.
2.3.2 Platform issues:

If a developer plans to develop in a multi-platform environment, then CORBA is the solution, for it is well known for its high compatibility and robustness over multiple platform networks (especially non Microsoft dependent environments). Finally, one is obviously advised to use DCOM if the target platforms are Microsoft based (Win9x/NT/2000/Me/XP), because DCOM is built into most of Microsoft’s recent OS releases and it is well known that it is the easiest, best and most reliable solution for those applications.

2.3.3 Network protocol:

Network protocol variability is a common issue in networking and internet applications. CORBA supports TCP/IP and has capability to plug in other protocols. On the other hand, DCOM was built with the internet in mind so it supports TCP/IP, NETBEUI (Microsoft windows Networking protocol), HTTP using COM internet services (CIS) and has support for plugging-in other protocols.

2.3.4 Security:

It is obvious that security of a system relies in the first place on the security capability of the platform it is running on. Various platforms have implemented their own ways of handling security issues, but there are further security mechanisms that should control access to distributed objects. Furthermore, the most profound architectures even allow for specialized security software plugins (as kerberos) to be used in their systems which must be considered as an advantage in those systems.

2.3.5 Internet scalability (IIOP):

As mentioned before, the point that puts DCOM to the scenes is its standardized binary interface, which makes communications over the
Internet easier. CORBA being an open architecture has many implementations with difficulty in communicating over the Internet and needs an Internet InterORB Protocol (IIOP) to link between 2 CORBA implementations. Nowadays, Internet connectivity is no more an option, so a distributed architecture is expected to have built in support for Internet connectivity protocols.

2.3.6 Fault tolerance:

Fault tolerance is an indispensable subject in modern distributed environments and there are several efforts and implementations found concerning the subject, each in its own way and unless a high availability guaranteed system is needed, in which case one may implement his own fault tolerance mechanism, he should base the choice according to the previously mentioned points.

2.3.7 Load balancing:

Load balancing is almost always implemented in modern distributed environments. However Microsoft’s support for multiple instances of the same component on different nodes and the COM capability to distribute load to several of these, may be more elaborate than that of the current CORBA implementations.

2.3.8 Interoperability with existing systems:

In the efforts of many object oriented distributed computing programmers, there were multiple trials to make bridges (translators) between the CORBA and DCOM ORB interfaces. By this time, there are many available, but none of them can be described as stable and reliable. Still the efforts are continuing to unite the advantages of the 2 worlds to get out the most of both.
2.4 Summary:

In this chapter we presented an overview of the development of distributed systems. We also emphasized on the development of various IPC mechanisms and the efforts to enhance and standardize them. Finally, we presented the issues that concern developers and vendors of distributed architectures and how different vendors target such issues.
CHAPTER 3

OVERVIEW OF SOME MODERN DISTRIBUTED OBJECT COMPUTING MODELS

3.1 Interoperation Layering Model:

Component technology depends on interoperation. Box [3] evaluated the degrees of interoperation according to the following layering model:

3.1.1 In-Memory Interoperation

The most intimate degree of interoperation can be obtained by mixing multiple components in memory. Component technology can offer excellent performance by standardizing an in-memory representation that all components must adhere to. Moreover, standardizing in-memory representation allows the supporting run time to offer various component management services with lower performance cost than would otherwise be possible.

COM standardizes the in-memory representation of object references based on simple C++-style virtual function tables, which makes in-process COM very easy to support on any platform [3]. Java standardizes the representation of component code and each virtual machine has a unique in-memory representation for objects. This approach does not restrict each virtual-machine implementation to innovation while still adhering to a common component format. However, this way, components must run in the same virtual machine to interoperate, which in the presence of versioning is not always possible. The CORBA specification depends on in-memory representation owing to the fact that the original goal of CORBA was to provide an object-based remote procedure call (RPC) system (Table 1).
3.1.2 Source Code Interoperation

Component technologies require the developer to program against a standardized application programming interface (API) of some sort for accessing component services. This way, a programmer can produce component source code that can be recompiled against another vendor's implementation of the technology.

COM exposes its services via the COM library and the Co APIs [3]. A significant subset of the Co APIs is consistent across platforms (Windows NT, Windows 95, Solaris, and Linux) and allows COM source code to be recompiled on multiple platforms. On the other hand, the CORBA specification defines a set of standard interfaces to be supported in any vendor implementation of the ORB in order to be considered CORBA compliant. This set of interfaces is considered a bare minimum, and can be further augmented by ORB vendors with proprietary extensions. Most Java-based component services are simply integrated into the language and don't necessarily have an explicit API. As a result, Java component services are fairly transparent. However, Java critics refer to the fact that one must port all of his software to the Java programming language, restricting the entire source-code base to the Java technology (Table 1).

3.1.3 Type Information Interoperation

Components should be well described to programmers who will utilize the component and to the underlying component system in order to ensure proper integration. All previously mentioned component technologies provide a standardized way of describing type information for utilization by developers and the supporting component architecture.

CORBA provides a text-based interface definition language (IDL) that allows objects to be described in a programming language-neutral manner [3]. All
publicly accessible data types are defined in IDL files that make it possible to access a CORBA object from any programming language that has ORB support. CORBA IDL is required to integrate with most CORBA products. COM, on the other hand, has a text-based IDL that resembles CORBA IDL (COM IDL supports more data types, CORBA IDL is easier to author and parse). The drawback to both COM and CORBA IDLs is that they tend to be good for authoring but not as good for interoperation and interchange. Owing to the complexity and richness of the IDL language structure, the IDL tools have a tedious function of parsing a rich language structure that has some dependencies on the C processor.

Microsoft has moved a step forward in solving this problem by providing a binary form of type information called type libraries for COM. Type libraries contain most (but not all) of the information in a COM IDL file in a representation that is easily understandable by a system-provided type library parser [3]. As Java components adhere to a standard self-describing class file format, no additional type information support is needed (Table 1).

### 3.1.4 Wire Interoperation

Components are the building blocks for building distributed applications. Accordingly, component technologies often define new network for communication between components across host machines.

Since, Windows NT relies heavily on the Open Software Foundation's Distributed Computing Environment (DCE) RPC mechanism, COM adopts the DCE RPC protocol for framing and transport, and uses the Network Data Representation (NDR) for parameter encoding. The Distributed COM (DCOM) protocol defines several proprietary DCE RPC interfaces to implement object functionalities within the system.
CORBA supports a variety of protocols with Internet Inter-ORB Protocol (IIOP) being the most common protocol for interoperation [3]. IIOP implements simple framing over TCP and uses the common data representation (CDR) for parameter encoding. Java supports both worlds, its native remote method invocation (RMI) protocol JRMP and IIOP/CDR. JRMP is based loosely on the Java-serialization format and can work over ordinary TCP or HTTP (Table 1).

A large amount of run-time support is needed for the previously mentioned network protocols used by COM, CORBA and Java to function properly. In the mean time, the Hypertext Transfer Protocol (HTTP) presented as the dominant Internet protocol. HTTP is simple, text-based, and requires very little run-time support to work properly and that’s what gives it its success on the Internet. Also, firewalls tend to block DCOM and CORBA traffic, while allowing HTTP packets into their secured networks.
<table>
<thead>
<tr>
<th>Interoperation Type</th>
<th>XML</th>
<th>COM</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>In-Memory Interoperation</td>
<td>W3C DOM (recommendation only), Simple API for XML (SAX), etc.</td>
<td>The COM API</td>
<td>The Java Programming Language</td>
</tr>
<tr>
<td>Text-based Type Information Interoperation</td>
<td>DTDs (legacy) XML Schemas/XML Data (future)</td>
<td>COM IDL</td>
<td>The Java Programming Language</td>
</tr>
<tr>
<td>Binary Type Information Interoperation</td>
<td>Same as text-based type info</td>
<td>Type Libraries</td>
<td>.class files</td>
</tr>
<tr>
<td>API-level Type Information Interoperation</td>
<td>None For DTDs; DTD replacement will just be XML, so any XML parser will work</td>
<td>LoadTypeLib, ItypeLib, etc.</td>
<td>java.lang.reflect</td>
</tr>
<tr>
<td>Wire Interoperation</td>
<td>XML (over HTTP, raw TCP, or message-based protocols)</td>
<td>DCOM (DCE based) over raw TCP, SPX, etc.</td>
<td>RMI/JRMP or RMI/IIOP or RMI/HTTP</td>
</tr>
</tbody>
</table>

*Table (1) XML and Component Integration Technologies [3]*
3.2 MICROSOFT COMPONENT OBJECT MODEL (COM+)

One of Microsoft's goals in developing COM+ has been to offer companies the benefits of multi-tier applications while hiding as much of the inherent complexity as possible [23]. The first version of COM shipped in 1993. Since that time, COM has grown from to become the core of Microsoft's multi-tier strategy.

3.2.1 The Foundation: COM

Microsoft established its multi-tier technology on the Component Object Model (COM). Although, it introduces several benefits, it was built on a complex technology that hides multiple low level details. The main theme of COM was distribution of object classes as binary coded components which meant that any software adhering to the COM interface model can interact with COM objects without dependency on the source code. This way, developers can offer their components in binary form without risking exposing their source code or model design. Also, this strategy reduced the hassle of compile time problems when any change in a component-implied recompilation of the whole system or at least part of it. The old development style produced a huge monolithic executable and necessitated a full compilation of the application with the change of one line of code.

COM was built upon the notion of Object Oriented Programming which implied that COM is used through instantiating COM objects from classes [23]. This resulted in better reuse and maintenance strategies. COM classes and clients exist in separate binary files and are able to bind at runtime using the COM infrastructure. One way that COM is similar to Java is that both provide a runtime dynamic loading mechanism which serves to instantiate objects of classes defined in binary files at runtime.
As COM provides definite interfaces, there are several COM enabled languages present nowadays. This allows a programmer to choose from a set of available languages to achieve a certain task while interoperating with other system components built in other COM-enabled languages by other team members. A list of COM-enable languages may include C++, Visual Basic, Java, Delphi and others.

Interface-based programming is used by many of today’s object oriented languages such as Java and C++. It is based on decoupling the interface from the implementation. The interface describes the publicly available methods for a class while the implementation part describes the way to execute them.

3.2.2 Distributed COM (DCOM:)

As COM was built essentially to overcome the inter-process boundary, it was considered a form of IPC mechanism. With the introduction of Microsoft Windows NT 4.0, COM proved to be more than an IPC mechanism. A new wire protocol was added for COM to allow it to extend across multiple nodes in a LAN environment [23]. Accordingly, COM now supports object communication in the same process, in different processes on the same node and in different process on different nodes, hence the name DCOM (Figure 6).
3.2.3 COM And MTS:

Microsoft Transaction Server (MTS) is an additional service that was released for Windows NT platform to run and control transactions from the middle-tier COM objects [23]. Actually, MTS was more than just a transaction monitor; it provided a runtime environment for COM objects that supported distributed transactions, integrated security and thread pooling with enhancements in the configuration and administration facilities. MTS also provided higher levels of scalability as it can share threads across multiple clients when the number of
clients exceeds a predefined threshold. This way it provides better runtime environment for applications that need concurrency support.

3.2.4 COM+:

With further developments and enhancements COM and MTS had to be unified into a single runtime with all the capabilities and facilities of both. Therefore, with the release of Microsoft Windows 2000, COM+ was a part of the default installation [23]. COM+ combined the benefits of both the COM runtime and MTS and provided more facilities. COM+ components can also be upgraded and enhance during the development cycle without affecting client applications and can transcend computer boundaries in a networked environment. COM+ also supports transactions, integrated security, thread pooling and offers other enhancements and services as object pooling, Queued components and COM+ events (see later).

3.2.5 COM+ Services:

COM+ and Windows 2000 include several built-in services that add to the runtime and are important to programmers of multi-tier applications.

3.2.5.1 Object Pooling:

Object pooling is a strategy by which COM+ pools objects waiting for client requests. Pools are configured and maintained on a per-component basis [9]. A pool consists of objects of a given CLSID. The pool will be populated to the minimum level previously defined, as long as object creation succeeds. As client requests for the component arrive, they are satisfied on a first-come first-served basis from the pool. If no pooled objects are available, and the pool is not yet at its specified maximum level, a new object is created and activated for the client.
If the pool reaches the maximum predefined level, further requests are queued, and objects are served according to availability from the pool. The number of objects activated and deactivated, should never exceed the maximum pool value. How long a client will wait can be controlled by timing out object requests after a specified period.

### 3.2.5.2 Internet Information Services

COM+ supports integration with ASP allowing for creation and running of business objects from an ASP page. In this way, a developer can distribute server logic to COM+ components while accessing it with scripts in ASP pages.

### 3.2.5.3 Microsoft Message Queue Service

Microsoft Message Queue Service (MSMQ) is an additional part to the platform services. MSMQ is a middleware service that facilitates messaging between various processes in a multi-tier application [23]. Messaging offers asynchronous and connectionless communication, not available with RPC and HTTP. MSMQ is a based on delivering messages to named queues asynchronously. Messages represent procedure calls between a client and a server with the facility that either party can do its work in the absence of the other. The main difference between a message and an RPC call is the a message is sent only in one direction while an RPC call is sent to a server and a client waits for the result of this call which can be viewed as bi-directional.

MSMQ can be of greater use to laptop users computers, who are constantly disconnecting from and reconnecting to the network. MSMQ allows application developers to create client applications that send messages to a queue on the network. If a laptop computer is offline, MSMQ automatically stores messages in a temporary local queue and when the laptop reconnects to the network, MSMQ,
sensing that the laptop is online again, automatically forwards the cached messages to the appropriate destination queue.

### 3.2.5.4 Queued Components

Queued components is another service of COM+ that allows a programmer to make use of MSMQ without having to program the MSMQ API [23]. This service is built on top of MSMQ and allows a developer to author queued components in the same way he develops ordinary COM+ components. The only limitation here is that methods cannot have output parameters or return values. For developing a queued component, there is an attribute that must be configured which tells COM+ that this component is to be queued. The COM+ application must also be configure to be queued and to be a listener. As a result, the Queued Components service automatically creates a special queue for the application and sets up a listener to handle incoming messages as they arrive. Client applications can start using a queued component once it is configured on the server. However, a client application creates a proxy object called a recorder instead of directly instantiating a queued component. This client-side proxy is identical to the desired object from the client’s view. The recorder has the function of recording method invocations from the client in MSMQ which are then transported over the network to the node where the queued component resides. The actual component is now instantiated by the queued components services and does the actual work for the client.

### 3.2.5.5 COM+ Events Service

There are times when applications require to receive notifications of critical events that take place in other parts of the system. The COM+ Events Services provide such a facility. It provides a service for delivering event notifications to system components [23]. Applications that send event notifications are called
publishers while those that receive them are called subscribers. Since publishers and subscribers are not supposed to know of the existence of each other, such events are called loosely coupled events. Events are defined inside event classes that applications know of. This way no modifications are needed to publishers when addition or removal of subscribers is required and similarly no modifications are needed to subscribers when addition or removal of publishers is required.

3.3 OBJECT MANAGEMENT GROUP COMMON OBJECT REQUEST BROKER (CORBA):

CORBA, the Common Object Request Broker Adapter, is a distributed standard developed by members of the Object Management Group (OMG) and their corporate members and sponsors. The first versions of CORBA were developed long before Java was publicized by Sun. The CORBA 1.1 specification was released in 1991 [13]. CORBA is a generic framework for building distributed object systems. The framework is a platform and language independent such that client stub interfaces can be specified in any programming language. The stubs and skeletons for objects must conform to the specifications of the CORBA standard in order for any CORBA client to access CORBA objects. The framework consists of the following elements:

3.3.1 The Object Request Broker (ORB):

The ORB is the core of the CORBA model for distributed objects. It provides the means of communication between clients and servers, so it should be running on client and server nodes to make this communication possible. At the client side, it accepts client requests for a remote object and finds its implementation in the system. It then routes the client requests to the remote object and waits for results to come back. At the server side, the ORB allows the registration of new objects by object servers. It receives the request from the client ORB, and uses the object’s skeleton interface to invoke the object’s activation method. The server ORB generates an object reference for the new object and
sends this reference back to the client. The client ORB is responsible for converting the reference to a language specific (C++, java, …etc) stub that the client uses to invoke methods on the remote object. As the client invokes a method on the required object, the client ORB marshals the parameters of the call to the server ORB which receives the request and calls the method on the object implementation through its skeleton interface. The result is marshaled by the server ORB and sent back to the client ORB where they are unmarshaled and delivered to the client program.

3.3.2 The Interface Definition Language (IDL):

Objects in a CORBA system provide interfaces describing the methods an object is capable of performing and how to call them. These interfaces are the means by which these objects communicate with each other. The IDL provides a platform and implementation independent way to define such interfaces. The IDL Language bares a lot of resemblance with C++ in terms of defining classes and their methods. However, IDL requires more specific information about objects interfaces like which arguments are input only, output only or input/output. An IDL interface is compiled into a client stub and a server skeleton and the input/output specifiers on method arguments are used to generate the code to marshal and unmarshal method arguments correctly (Figure 7).

3.3.3 The Communication Protocol:

This is a binary protocol for communication between ORBs, called the Internet InterORB Protocol (IIOP).

Earlier, the CORBA standard did not include a low-level binary specification for the inter-ORB network protocol. Instead, it described the protocol in generic terms that a compliant system had to implement [13]. However, this resulted in a mess, as vendors were implementing CORBA object servers that couldn’t communicate with each other, even though they followed the standard until the IIOP was specified in the 2.0 release of the CORBA specification.
3.3.4 Server Implementations:

An IDL interface for a class needs to be compiled into a server skeleton and a client stub. For this reason, IDL translators (compilers) exist for C, C++, Smalltalk, Ada, Java and other common languages [13]. Stubs and skeletons need not be compiled into the same programming language. Server implementations are built by compiling the IDL interface into a native language interface and an implementation skeleton. Then, the implementation of the object is provided by deriving from the skeleton and writing the implementations for the methods on the object interface. When the implementation is defined for the object, registration of the object implementation with the server ORB renders the object ready for use by clients. Also, registration may take place with the CORBA Naming Service which allows clients to access it by name (Figure 7).

3.3.5 Client Stubs:

Clients use a stub to access the data and methods on the remote instance of the object. The same IDL interface used to generate the server skeleton is now used to generate the client stub using an IDL compiler. The stub uses CORBA specific methods to marshal method arguments and send them to the server and to unmarshal return values and output parameters [13]. If a client requests a remote object reference, it is given the reference in the form of a stub instance. The client can get a connection to a remote object by means of the ORB which should be running on the client node. The ORB should be supplied with the remote host address and port to communicate with. Once communication is established, requests can be sent through the ORB’s Naming Service to ask for a remote object by name. The client ORB makes a connection to the server ORB and asks for the named object. The client ORB creates a reference to the requested object as an instance of the stub generated from the IDL interface. The client can begin invoking methods on the stub interface which are routed by the client ORB to the server ORB. The method calls are then executed on the server object.
implementation and the results are marshaled by the server ORB to the client ORB through the stub to the client (Figure 7).

![CORBA Communication Scenario](image)

*Figure (7) CORBA Communication Scenario. The Client request passes through the CORBA runtime (ORB) which marshals the parameters and pass them to the network to be received by the ORB on the next node and unmarshaled to the server to process the request. It then sends the result back the reverse way round. [27]*

### 3.4 Java Remote Method Invocation (RMI):

The Java Remote Method Invocation (RMI) package is a Java-centric scheme for distributed objects. It is now part of the core Java API. Java RMI offers most of the critical elements of a distributed object system for Java, with some additional features made possible by the fact that RMI is a Java-only system [14]. RMI’s object communication facilities are similar to CORBA’s IIOP, and its object serialization feature provides a way to transfer or request object streams from one remote process to another.
3.4.1 Remote Object Interfaces:

Since RMI is a Java-only distributed object scheme, all object interfaces are written in Java. Client stubs and server skeletons are generated from this interface in a way similar to but slightly different from CORBA. The interface for the remote object is written extending the java.rmi.Remote interface. The Remote interface only servers to identify remote objects to the RMI system. One of the disadvantages of RMI is that an existing interface has to be modified in order to apply it to a distributed environment.

3.4.2 Server Implementations:

A server implementation interface has to be written for the defined object interface. The server implementation extends the java.rmi.UnicastRemoteObject class and implements the interface of the object. The UnicastRemoteObject class is an extension of the RemoteServer class, which acts as a base class for server implementations of object distribution schemes as replicated objects, multicast objects, or point-to-point communications.

3.4.3 The RMI Registry:

In RMI, the RMI registry assumes the role of the ORB and Naming Service in CORBA. The registry runs in its own Java runtime environment on the host that’s serving objects. Unlike CORBA, the RMI registry is only required to be running on the server of a remote object [14]. Clients of the object use classes in the RMI package to communicate with the remote registry and look up objects on the server. The RMI registry should be started on a host by running rmiregistry command. By default the registry listens to port 1099 on the local host for connections, but any available port can be specified. Object implementations are registered by name using java.rmi.Naming interface after being executed. Finally, registered classes can be located by a client using the lookup() method on the Naming interface which returns an object reference to be used afterwards.
**3.4.4 Client Stubs and Server Skeletons:**

Following interface definition and a server implementation, a client stub and a server skeleton can be created for this object using the RMI compiler (*rmic*). The interface and the implementation are compiled into byte codes like normal classes. Then the stub compiler has the role to create the client stub and a server skeleton from the same interface file. A client stub is returned to a client when a remote instance of the class is requested through the Naming interface. The stub has internal links to the object serialization subsystem in RMI to be able to marshal and demarshal method parameters and return values. The server skeleton acts as an interface between the RMI registry and instances of the object implementation residing on a host. As a response to a client request for a method invocation, the skeleton is called to extract the serialized parameters and pass them to the object implementation.

**3.4.5 Object Serialization:**

Object serialization is one of the advantages Java RMI has over other distributed computing technologies [14]. The *java.io* package includes classes that can convert an object into a stream of bytes and reconstruct the stream into an identical copy of the original object on another host. An object that implements the *java.io.Serializable* interface in one process can be serialized and transmitted over a network connection to another process on a remote host. The object can then be reconstructed on the remote host from the received stream.

**3.5 ENTERPRISE JAVA BEANS (EJB):**

Enterprise Java Beans is a distributed, transactional, server-side component model [16]. It concentrates more on writing business logic instead of writing server-side system code. In the next few sections we introduce the features of EJBs as a distributed component model.
3.5.1 Services Framework

Enterprise Java Beans is a specification which allows vendors to create EJB server implementations from the Specification [16]. This allows many vendors to implement their own server-side products to provide EJB services and gives the developer the freedom to choose among several EJB implementations. The Specification describes several details involving vital services such as transactions, security, persistence and naming. It does not specify implementation topologies and thus allows vendors to provide enhancements without sacrificing portability regarding core services. Vendors can provide advance features like database caching, resource pooling, resources sharing, fail over, clustering, and advance distributed transactions which are not described in the core Specification requirements as much as they meet the interface and semantic requirements of the Specification.

Enterprise Java Beans specification describes a Java component model which details a services framework where components can be portably deployed [16]. Enterprise beans do not typically send or receive intra-process events nor is there mention of properties. Contrary to other models as COM+, customization is not performed at development time using properties, but at runtime (deployment time, actually) using a deployment descriptor. Java has many component models including Applets, Servlets, and JSP TagLibs. Applets typically execute in the context of a browser. Servlets execute in the context of a web server (or an application server). Enterprise Java Beans focus on distributed, inter-process communication and shares other Java component models the fact that components execute only in the context of their container which is the EJB server, which can be embedded in an application server, transaction server, middleware integration server, database server, etc.
3.5.2 EJB Architecture:

The EJB server is the high-level process or application that manages EJB containers as well as providing access to system services. The EJB server may also provide enhancements and features added by several vendors as optimized database access interfaces and availability of CORBA services. According to the specification, an EJB server is required to provide a JNDI-accessible naming service and a transaction service. The EJB container is an abstract concept and set of functionalities. Developers create enterprise beans that are deployed to the EJB server and contained in an EJB container within the EJB server [16]. The structure of an enterprise bean includes the home interface, the remote interface, the implementation (EJB class) and the deployment descriptor. The home interface provides life cycle management and location services, the remote interface defines the interface of business methods for remote clients and the EJB class contains the implementation of business logic. Enterprise Java Beans 2.0 added the notion of a local enterprise bean. A local bean has a local interface and a local home interface. Before this, the clients typically interfaced with Enterprise Java Beans via some form of remote method invocation (RMI) (Figure 8).

3.5.2.1 The Local Interface:

The local interface provides the interface for the business methods. Local beans can only be accessed locally by other enterprise beans and web components but not by remote clients. The EJB container contains enterprise beans and manages one or more EJB classes and/or instances. It is also responsible for providing services like transaction control, lifecycle management, and security to the contained bean. The container is not visible to the client or to the contained bean. The container acts by intercepting method invocations made on the bean, and providing services to the bean transparently. An EJB can be defined to allow container managed transactions and a certain method can be defined to require a transaction. All of this can be specified in the deployment descriptor and
accordingly, the container starts a transaction context if the specified method is invoked.

3.5.2.2 The Home Interface:

The home interface is the interface for managing the lifecycle of enterprise beans and finding enterprise beans [16]. The home interface lists the methods for creating, locating, and removing instances of EJB classes. The developer of the EJB class must define the home interface while the container vendor generates the home object implementation from the home interface. The remote interface lists the business methods in the EJB class. The EJB Object implements the remote interface, and is the object that the client must use to access the business methods of the EJB instance. The EJB Object and home object are considered part of the container. The only difference between the local and the remote interfaces is that the local interface is not a distributed object accessible from remote clients. The client never gets a reference to the EJB instance, only its EJB Object instance, which it accesses through the interface. The EJB Object receives requests from clients and delegates them to the EJB instance which provides any necessary wrapper functionality in the process. The client is an application that uses the home object to locate, create, or destroy instances of an EJB class, and uses the EJB Object to invoke the business methods of an instance. While remote clients usually use Java RMI over IIOP to access the home object and EJB Object, the server can provide whatever form of RMI to communicate.

3.5.3 Benefits of Enterprise Java Beans

Several considerations are to be studied when planning to build a distributed application using EJB. For many applications the advantages far outweigh the disadvantages, especially for more complex applications.
3.5.3.1 Establishing Roles for Application Development:

A developer does not have to worry about complex things as resource sharing, explicit transactions, security, connection pooling, and thread synchronization. Developers build their application on the application server and the EJB server vendors will take care of providing support for complex services, and make them available to the enterprise bean. The deployer can take care of installation issues in a simple and portable fashion.

![Figure (8) The J2EE Object Model](image)

3.5.3.2 Component marketplace:

Developers can buy off the shelf components and assemble them with their own components into enterprise applications. Developers and application assemblers can edit deployment descriptors to modify several environment settings.
to customize them for their enterprise application. Furthermore, application assemblers and developers can customize the components to work with their database infrastructure by modifying deployment descriptors.

3.5.3.3 Automatic Transaction Management:

One of the services transparently provided by the container vendor is transaction control. The person writing the business functions does not have to worry about starting and terminating transactions. The bean developer specifies the transactions in the deployment descriptor which are implicitly executed at runtime. Thus, components can be assembled into another enterprise application with different transaction needs and developers do not have to be aware of transaction details.

3.5.3.4 Distributed Transaction Support:

Distributed transaction support provides part of transaction transparency. A distributed transaction is a transaction that access processes on remote servers. This allows beans on different servers to participate in the same transaction. A client can start a transaction and then invoke methods on beans in two different servers. Methods in one bean can call methods in another bean while executing in the same transaction context.

3.5.3.5 Portability:

EJB specifications were designed to provide an environment where components can be written once and portably deployed into any EJB server. The Java2 Enterprise Edition (J2EE) Specification provides a clear specification of the requirements for a Java server. If EJB server is J2EE compliant then beans written for it can be deployed to other EJB servers. It is the developer’s concern to understand what J2EE and EJB provide, and to be careful when using additional functionality.
3.5.3.6 Scalability and Robustness:

Although the architecture of EJB may appear to be complex, the Specification was written to allow vendors to provide extremely high-performance implementations. Moreover, features as load balancing; data caching, clustering and fail over have been implemented in many high-end EJB servers to obtain scalable and robust frameworks.

3.5.3.7 Integration with CORBA:

In many ways CORBA and EJB are natural complements to each other, but in other ways, EJB and J2EE replaced the need for many CORBA-services [16]. The use of IIOP allows CORBA clients to access enterprise beans as EJB clients. Thus, for example, C++ clients can access enterprise beans written in Java. CORBA services provide a wealth of features to an application developer. Instead of trying to replace these services, many EJB server vendors provide access to CORBA services into their product and provide access to these services through JNDI, and standard Java APIs like JMS. In addition, J2EE and EJBs provide support for bean developers to use CORBA services without needing to become experts in CORBA. An example of supporting CORBA services is the transaction support provided by EJB servers to enterprise beans. The implementation of the distributed transaction service is CORBA OTS. Its transaction and naming services will have to support the CORBA OTS and Naming Service interfaces, respectively, to provide full interoperability with CORBA clients and servers. CORBA and CORBA Services are very difficult to use compared to J2EE equivalents. Conversely, RMI over IIOP provides the ability to provide CORBA services for Java without learning CORBA Interface Definition Language (IDL). J2EE 1.3 uses CORBA IIOP; thus, making CORBA-IIOP the robust distributed object protocol of the Internet. Moreover, CORBA ORBs are readily available because Java Standard Edition, v1.2 and higher includes a CORBA ORB. J2EE and EJB do not
replace CORBA but renders it a lower level protocol, just like TCP/IP is to Ethernet.

3.5.3.8 Vendor Enhancements:

The real value in Enterprise Java Beans is the flexibility the specification allows for vendors to provide their own enhancements. Features like automatic object-relational mapping when using container-managed persistence, gateway services into existing applications, customizable business frameworks, and integration of CORBA services are just a few examples of added value that may be provided by vendors.

3.6 MICROSOFT SIMPLE OBJECT ACCESS PROTOCOL (SOAP):

SOAP is a simple and lightweight mechanism for exchanging structured and typed information between peers in a decentralized, distributed environment using XML [4]. SOAP does not define any application specifications, programming model or implementation topologies; it defines a simple modular packaging model and encoding mechanisms for encoding data within modules. Hence, SOAP can be used in a large variety of systems ranging from messaging systems to RPC.

SOAP consists of three parts:

?? **The SOAP envelope:** describes what is in a message; who should deal with it, and whether it is optional or mandatory.

?? **The SOAP encoding rules:** define a serialization mechanism that can be used to exchange application-defined data types.

?? **The SOAP RPC representation:** defines a convention that can be used to represent remote procedure calls and responses.
SOAP was designed for simplicity and extensibility [4]. There are several features from traditional messaging systems and distributed object systems that are not part of the core SOAP specification.

3.6.1 The SOAP Message Exchange Model

Essentially, SOAP messages are one-way streams from a sender to a receiver; however, SOAP messages can be combined to implement request/response patterns. SOAP implementations can be optimized to utilize the specific characteristics of a particular network system. The HTTP binding described later provides for SOAP response messages to be delivered as HTTP responses, using the same connection used for the request. Regardless of the protocol, messages are routed along a so-called "message path". This allows for processing at one or more intermediate nodes in addition to the ultimate destination.

3.6.2 Relation to XML:

SOAP messages are encoded using XML. According to Microsoft, SOAP defines two namespaces:

?? The SOAP envelope namespace identifier:

"http://schemas.xmlsoap.org/soap/envelope/

?? The SOAP serialization namespace identifier:

"http://schemas.xmlsoap.org/soap/encoding/

A SOAP message should not contain a Document Type Declaration or Processing Instructions. SOAP uses the local, unqualified "id" attribute of type "ID" to specify the unique identifier of an encoded element and uses the local, unqualified attribute "href" of type "uri-reference" to specify a reference to that value. It is also, generally permissible to have attributes and their values appear in XML instances or in schemas.
3.6.3 The SOAP Envelope:

A SOAP message is an XML document that consists of a mandatory SOAP envelope, an optional SOAP header, and a mandatory SOAP body [4]. This XML document is referred to as a SOAP message.

?? The Envelope: is the root element of the XML document representing the message.

?? The Header: is a generic mechanism for adding features to a SOAP message without prior agreement between the communicating parties. SOAP defines a few attributes that can be used to indicate recipients dealing with a feature and whether it is optional or mandatory.

?? The Body: is a container for core information of the message intended for the ultimate recipient. SOAP defines one element for the body, which is the Fault element used for reporting errors.

3.6.4 Envelope Versioning Model:

Rather than defining a traditional versioning model based on major and minor version numbers, SOAP treats the envelop element associated with the previously mentioned envelop namespace as a version. If a message is received by a SOAP application in which the SOAP Envelope element is associated with a different namespace, other than the default, the application must treat this as a version error and discard the message. If the message is received through a request/response protocol such as HTTP, the application should respond with a SOAP VersionMismatch fault code message.

3.6.5 SOAP Header

The Header element is encoded as the first immediate child element of the SOAP Envelope XML element. All child elements of the Header element are
called header entries. Extensions can be implemented as header entries such as authentication, transaction management, payment etc.

3.6.5.1 Use of Header Attributes

The SOAP Header attributes define the way a recipient of a SOAP message should process the message as described before. A SOAP application should generate a SOAP message using only the SOAP Header attributes on immediate child elements of the SOAP Header element. Similarly, a recipient of a SOAP message should ignore all SOAP Header attributes that are not applied to immediate children of the SOAP Header element.

An example is a header with an element identifier of "Transaction", a "mustUnderstand" value of "1", and a value of 5. This would be encoded as follows:

```xml
<SOAP-ENV:Header>
  <t:Transaction
    xmlns:t="some-URI" SOAP-ENV:mustUnderstand="1">
    5
  </t:Transaction>
</SOAP-ENV:Header>
```

3.6.6 Encoding Types in XML:

3.6.6.1 Simple Types:

SOAP adopts all the types found in the "XML Schema Part 2: Datatypes" Specification [46]. The data types declared in the XML Schema specification may be used directly in element schemas. Derived types may also be used. An example
of a schema fragment and corresponding instance data with elements of these types is:

```xml
<element name="age" type="int"/>
<element name="height" type="float"/>
<element name="displacement" type="negativeInteger"/>
<element name="color">
  <simpleType base="xsd:string">
    <enumeration value="Green"/>
    <enumeration value="Blue"/>
  </simpleType>
</element>

<age>45</age>
<height>5.9</height>
<displacement>-450</displacement>
<color>Blue</color>
```

3.6.6.2 Compound Values, Structs and References to Values:

Compound Values are represented in soap as accessor elements [4]. If an accessor element is distinguished by name, the accessor name is used as the element name. Accessors whose names are local to their containing types have unqualified element names; all others have qualified names.

The following is an example of a struct of type "Book":

```xml
<e:Book>
  <author>Henry Ford</author>
  <preface>Prefatory text</preface>
  <intro>This is a book.</intro>
</e:Book>
```
3.6.6.3 Arrays:

SOAP arrays are defined as having a type of "SOAP-ENC:Array". Arrays are represented as element values, with no specific constraint on the name of the containing element [4]. Arrays can contain elements of any type, including nested arrays or complex types. The representation of the value of an array is an ordered sequence of elements constituting the items of the array.

The following example is a schema fragment and an array containing integer array members.

```
<element name="myFavoriteNumbers"

type="SOAP-ENC:Array"/>
<myFavoriteNumbers

SOAP-ENC:arrayType="xsd:int[2]">
  <number>3</number>
  <number>4</number>
</myFavoriteNumbers>
```
3.6.7 Using SOAP with HTTP:

The SOAP/HTTP combination provides the advantage of being able to use the decentralized flexibility of SOAP with the rich feature set of HTTP \[4\]. Transferring SOAP through HTTP does not mean that SOAP overrides existing semantics of HTTP but rather that the SOAP over HTTP adds to HTTP semantics.

SOAP naturally follows the HTTP request/response message model providing SOAP request parameters in a HTTP request and SOAP response parameters in a HTTP response.

HTTP applications must use the media type "text/xml" when including SOAP entity bodies in HTTP messages.

3.6.7.1 SOAP HTTP Request:

The SOAP/HTTP request binding only defines SOAP within HTTP POST requests. The SOAP Action HTTP request header field can be used to indicate the intent of the SOAP HTTP request. The value is a URI identifying the request intent. SOAP places no restrictions on the format or specificity of the URI or that it is resolvable. An HTTP client must use this header field when issuing a SOAP HTTP Request. The presence and content of the SOAPAction header field is a means by which firewalls can appropriately filter SOAP request messages in HTTP. The header field value of empty string (""") means that the intent of the SOAP message is provided by the HTTP Request-URI.

3.6.7.2 SOAP HTTP Response:

SOAP HTTP adopts the structure of the HTTP Status codes for communicating status information in HTTP. In case of a SOAP error while processing the request, the SOAP HTTP server must issue an HTTP 500 "Internal
Server Error" response and include a SOAP message in the response containing a SOAP Fault element indicating the SOAP processing error.

Example SOAP HTTP Using POST

POST /StockQuote HTTP/1.1
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "http://electrocommerce.org/abc#MyMessage"

<SOAP-ENV:Envelope...

HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
<SOAP-ENV:Envelope...
3.7 Summary:

In this chapter, a synopsis of current distributed object computing technologies has been given, showing the strengths, drawbacks and modes of interoperability. The choice of a system depends greatly on the nature of problem at hand, platform, availability of facilities, programming language and other factors as well. Following is a table that summarizes the features of the previously discussed technologies and points out the major differences that may recommend one system over another in a certain problem environment.
<table>
<thead>
<tr>
<th>Distributed Model</th>
<th>CORBA</th>
<th>COM</th>
<th>JAVA RMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Platform</td>
<td>Any platform that has a CORBA implementation</td>
<td>Microsoft Platforms</td>
<td>Any platform with a Java VM</td>
</tr>
<tr>
<td>Programming Language</td>
<td>Any language for which there is an IDL compiler</td>
<td>Any language that supports COM</td>
<td>JAVA only</td>
</tr>
<tr>
<td>Network Protocol</td>
<td>TCP/IP, SPX, NETBEUI</td>
<td>TCP/IP, SPX, NETBEUI</td>
<td>TCP/IP</td>
</tr>
<tr>
<td>Transfer Protocol</td>
<td>InterORB Protocol</td>
<td>DCE RPC compliant</td>
<td>Proprietary</td>
</tr>
<tr>
<td>Interface Language</td>
<td>IDL, DII</td>
<td>COM IDL (More data types), Binary Type Libraries</td>
<td>Built-in Java Interfaces</td>
</tr>
<tr>
<td>Security</td>
<td>Built-in, Pluggable</td>
<td>Built-in, Pluggable</td>
<td>Built-in</td>
</tr>
<tr>
<td>Complexity</td>
<td>+++</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>Interoperability</td>
<td>IIOP, CORBA-DCOM Bridges</td>
<td>CORBA-DCOM Bridges</td>
<td>JNI for C/C++ Clients</td>
</tr>
<tr>
<td>Object Serialization</td>
<td>By reference</td>
<td>By reference</td>
<td>By Reference and Value</td>
</tr>
</tbody>
</table>

*Table (2) Comparison Between Different Distributed Object Technologies*
CHAPTER 4

THE PROPOSED MODEL

4.1 Introduction:

As we reviewed the most popular modern distributed object frameworks present nowadays, we have come across several strengths and weaknesses for each. We can safely conclude that there is no such thing as the ultimate distributed object environment. Moreover, the results of our comparison points out that several factors play a crucial role in the choice of the suitable distributed model to use. These factors may include the size, complexity, implementation language, connecting to other systems and others.

The purpose of this work is to make use of XML technology to target the ultimate distributed model in an attempt to resolve the weaknesses and complexities of already existing distribution models. XML, not only suites making the inter-object communication protocol and core messaging framework, but can also be used in other aspects of a distributed system as registering components, implementing security, exposing object interfaces and more. Below are proposed solutions to some issues in distributed systems made possible through the use of XML.

4.2 Platform Preference:

As mentioned before, the choice of a platform for implementation had a drastic role in the choice of the distributed computing model to be used. The choice of Microsoft Windows directed the attention primarily towards COM and if it was required to connect to previously built
CORBA dependent systems this had to be through switching to CORBA or using a CORBA/COM bridge, which in either case showed inconvenience and/or complexity. On the other hand, Java introduced flexibility by being platform independent and has support for CORBA but eventually one has to stick to the Java language. Also, enterprise Java beans is becoming more and more popular, being CORBA compliant (only for CORBA clients) but also one has to stick to writing Java for implementing servers. Using XML to communicate between objects can be considered as using a universal language that everyone talks nowadays and that’s what Microsoft has built upon its new SOAP technology. Fortunately, SOAP is only a specification and developers are free to build their own implantation the way they like it, an advantage and a disadvantage at the same time. XML is used by every language and on any platform and puts no restrictions on the developer to use whatever language he prefers on whatever platform.

### 4.3 Programming Language:

Choosing a programming language for developing a distributed system had to be based upon the model of distribution chosen. Several factors had to be considered, support by the distribution model, the presence of IDL compilers and the presence of an implementation of the chosen language on the development platform. As an example, choosing COM as the distribution model, we have a wide choice of COM compliant applications on the Microsoft Windows platform, but it is difficult to incorporate UNIX machines into this system. Also choosing Java RMI as the distribution model, a developer is stuck to Java only programming regardless of the platform. Almost every known programming language now knows about XML and even has a standardized parser that uses SAX or DOM implementations. XML
parsers are present for COM compliant languages (Microsoft Visual Basic, C++, Microsoft Office), Java, C, C++, Perl, Python, PHP, Ada and others.

4.4 Complexity:

Components in modern distributed systems depend almost always on some kind of a Broker mechanism to regulate the process of remote invocation. Sometimes this Broker or runtime module gets too complex for providing methods to lookup the objects, identify their interfaces, check for security and marshal and demarshal method parameters. Also, referencing a remote object should involve the presence of a server skeleton and a client stub to perform the communication. These files had to be present before an object communication could take place, although some systems provide a runtime method for identifying an object’s interfaces (CORBA DII) [14].

By implementing an XML parsing capability in any component in the system, we can rid ourselves off the complexity of an Object request broker for marshalling and demarshalling method arguments, creating objects, serializing/deserializing objects and checking security. In this way, we can implement a simpler naming service component for object lookup and leave objects to communicate directly with each other which will increase performance and leave more time for the naming service to handle more requests. Some models implement the naming service as a separate component to relief the Broker from performing the lookup function (ex: the CORBA naming service), still it has to communicate with the client through the ORB runtime.
4.5 Security:

Implementing security was one of the complex issues in distributed systems and was built inside the ORB that uses a main dictionary for all the objects on the node and their security access rights. Microsoft COM uses the Windows Registry as a component database with all their access rights still forcing the user more and more to use a Windows platform. On the other hand, CORBA uses a more complex component database so as to maintain the non-platform dependent architecture. Java uses the RMI Registry to handle such issues.

Using XML we can easily implement security as local component specific XML documents which can be manipulated easily by helper programs and utilities or even, for more convenience, manually by an XML experienced system administrator or developer. This would enhance development time and increase debugging flexibility. Moreover, authorization had to go through the runtime module of the distributed model to give a client access to the server, however, using this technique, an object can be its own security guard. Actually security is divided into two levels, the first is the permission to instantiate an object which can be left to the runtime module or the Broker, and the second is the permission to access specific functionality or methods of the component itself which can be left to the component to handle. This way, the burden or overhead of security can be divided among the Broker and the components. On the other hand implementing security inside a component is not an overhead to the developer, it gives him the flexibility to add whatever security implementation he requires, or even not to implement it at all. After all, it’s just the matter of manipulating more XML documents.

Another security issue is that the transfer of ASCII XML documents across the network or over the internet may present a large security hole for sniffers, especially if there is highly confidential
information in the XML packets as passwords or transactional details. Fortunately, if security is a must, encryption is always there. An XML document can be encrypted at the sending side and decrypted at the receiving side in cases necessitating this overhead. This feature would also be left to the developer and would be implemented at the component level, just adding more to the functionality of the component. Figure (9) shows the authorization scheme used by Microsoft COM+ runtime.

**Figure (9) DCOM Authorization scenario.** Client requests are routed to the security provider which evaluates the client access rights through the windows registry and returns either a failure or success for the authorization process[25]

### 4.6 Serialization:

One of the most availing features of Java RMI is the Java Serialization interface which is deployed in Java distributed applications to pass Objects by value. This feature allows for the creation of remote objects and sending them to the client’s node to be reconstructed and used, and thus freeing the server side for handling more requests. Other distributed models use passing object by reference as a default. The
object is created at the server side and a reference to the object is passed to the client side to be used through the client stub. The runtime component marshalls call parameters to the server skeleton which hands it on to the server implementation to execute the call and then the other way round. Using XML, transferring an object to a serializable XML document makes this feature available to all programming languages that support XML. The hierarchy of an XML document aids in representing objects and embedding the details of object status and attributes in the document as well as other objects or collections it may contain. The transfer of text files is always easy and very efficient and that’s what made HTML/XML the web’s most popular document transfer format. An example of a serialized object looks like:

```xml
<OBJECT Class="group" Name="Users" Users="Alan, David, Ahmed, Sandy">
  <Users>
    <User Name="Alan" FullName="Alan Moskovik" Title="IT Manager" Age="35"/>
    .
    .
    .
    .
  </Users>
</OBJECT>
```

4.7 Compression:

As we progress more into the system, XML documents may get larger and larger which would eventually load the network and affect the transfer of packets. Compression is a natural solution, especially that XML documents are ASCII which would lead to a very effective compression ratio sometimes reaching 10:1. This would greatly accelerate the transfer rate and lessen the communication overhead in systems with
limited message size as UDP dependent systems (see later). However, as the proposed system implies, this has also to be built into the functionality of the system components, in addition to the previously mentioned features. Nevertheless, a wrapper library can be provided to wrap the whole building process of an XML message to the eventual structured, encrypted and compressed form. Even, if we use an unusual or a secure compression technique with a key for example then we might get the result just in one step.

4.8 Versioning:

As we have discussed XML before, we mentioned DTDs and XML schema structures, and as we discussed several distributed models we came over several issues concerning multiple versions of components and how this is handled. Implementing a new version of a component in the conventional way would imply writing a new interface and recompiling it into a client stub and server skeleton. More recent features of distributed models are trying to overcome this by providing a way to use multiple versions of a component. Such a feature, with the use of XML Schema or DTDs, can be implemented with the utmost flexibility. Imagine an RPC sent to a server with the instructions of how to deal with it. For example, if an RPC is sent to a server component, the arguments should be sent as specified by the interface regarding ordering and data type. With the proposed approach, we can send the arguments in any order and the server can figure out which parameter it needs and also which version of the call it is to process. Such a feature is implemented in some languages as named arguments (ex: Visual Basic 6.0). Also, the server can verify the syntax of a call using a prebuilt XML interface document and issue error messages if the call is badly formed.
4.9 Flexibility:

What we have mentioned so far can account for a great deal of flexibility. The proposed XML oriented model can eventually make rewriting and recompiling objects necessary only in major core changes but not trivial changes. Also interface definition language here (IDL) is replaced by XML documents and no IDL compilers are necessary, everything is based on XML parsers. This approach would give the developer the freedom to do so without restricting him to a predefined set of instructions to stick to when building a system. Also the proposed model is aiming towards minimizing the standards needed to build applications that can communicate with each other, on the other hand, it can put more burden on the developer to implement much of the features that are hardcoded inside the conventional models. Moreover, third party components, services and utilities can be readily implemented in any language and on any platform without the need to ask for a sophisticated standards documentation or a special API to join the system.

4.10 Interoperability and Standardization:

Finally, several legacy systems are already implemented using one or more of the previously discussed distributed models, and several attempts were made to bridge already established architectures using concepts as CORBA/DCOM bridges but up till now there hasn’t been much success in this area, which is by all means against nowadays standardization policy. The standardization of a calling system or convention would facilitate such interoperability to major software products. This has urged major software companies as Microsoft to use XML as a messaging protocol and to standardize this protocol for interoperability and thus introducing SOAP.
SOAP is beginning to gain popularity and support among software vendors, nevertheless, as we have seen, the SOAP structure is complex and is built primarily for the internet being dependent on the HTTP protocol. The proposed solution would have not much to adjust and even other software vendors may not have to change their software radically to implement it. They may not even have to make a new release, instead they can introduce just patches or plug-ins to support this calling convention. Moreover, a developer using the proposed model can benefit from all features of the new model and add just one component as a service to act as a bridge or a translator for another system. For example, using the Java language to build a bridging component that can understand RMI and XML would allow us to bridge the two systems together, and adding another component which can understand COM or CORBA leads us to have three systems acting in synergy. Finally, if we are talking about bridging and standardizing different software packages, then of course, we are solving the platform, language and architecture issues. It is very easily implemented on different hardware architectures, different platforms and using different languages with minimal effort.

4.11 Components:

The proposed system should have the following basic components. By convention, all system components should have the ability to create and parse XML documents.

4.11.1 Brokers (Runtime Components):

The runtime components are the components that are running all the time if the system is up. They represent different functionalities in different models but in this model they are always waiting for requests by clients to get object references. The Broker can have also the
functionality to authorize access to a certain component through looking up some XML access permissions files. Brokers on all nodes should be listening to the same predefined port as described later.

### 4.11.2 Clients:

Clients are components that request a reference to a server component and perform some operations using it. As previously noted a client should have the capability to encode RPC calls into XML messages and to decode XML messages to extract the results of the operation. The client components reserve a port on the node on which they start up and this port number is used to make a reference to them together with the IP address of the node.

### 5.11.3 Servers:

Servers are the components that have some functionality in the system and are waiting to serve some clients to perform some operation. Servers are usually instantiated by the Broker and can usually communicate with the client through the Broker or the runtime component, but in the proposed model we give the Broker the task of only instantiating the component and leaving the communication to the server and client components to handle. Also detailed access rights to several methods are left to the server component to decide using XML access permissions files. As before a server component reserves an empty port on the node it is instantiated on and the IP and port number together make the reference to the object.

### 4.12 Communication Channels:

As described by Bal et al [2], message passing can be through synchronous, asynchronous, rendezvous or RPC calls. Most of the previous distributed models are built upon the RPC methodology which
blocks the sender until the return parameters are received. The rendezvous mechanism resembles the RPC except that the sender does not block except until the receiver notifies receipt of the message. Synchronous message passing necessitates that the sender blocks until it receives a reply from the receiver and hence synchronizes with it, while asynchronous mode allows the sender to continue working after sending the message. In the last case, the sender has to find a way to identify replies from different sources and that is where message tagging plays a role as we shall see later.

The need for a simple and flexible communication protocol in such a system is a major issue. XML messages are simple ASCII messages and we have agreed upon giving the client and server a direct means of communication and not through the Broker or runtime module, so a simple IPC mechanism which supports networking should be used. This mechanism is obviously sockets and so the TCP/IP networking protocol is used. Although, some authors describe opening sockets and establishing a connection to be a tedious and code exhausting procedure, this coding technique tends to be standard and is a template to be used whenever a socket communication is required [21]. Wrapper classes, libraries, components are spread all over the developer community to support facilitating socket communication and ridding the developer to get down to the details of sockets. In case a developer needs more control to implement his own socket mechanism, he has the freedom to do it.

The most popular type of sockets used nowadays is the TCP sockets and UDP sockets. Any of the two socket types can be used in the model, depending on the requirements and the functionality of the system, as we shall see later. A system that depends on short messages
and heavy communication should use UDPs for performance, however a system with large message blocks and mild communication overhead can use TCP for more reliability of the communication. Mixing of the two techniques is possible in applications as required especially if the need for broadcasting and reliability exist for instance. Another thing worth to mention is that other IPC mechanisms that are more efficient than socket communication should be used when component intercommunication is required on the same node. The incurred overhead and security violation of opening a port for inter-process communication on the same node can be avoided by using a more appropriate IPC mechanism. This mechanism is variable according to the host operating system design. For simplicity, the current implementation uses sockets for all component communication and also to provide a homogenous communication environment for testing and analysis of the results.

4.12.1 Problems with UDP Sockets:

- **Reliability:**
  
  UDP sockets have a problem of reliability, that is, on delivery of the packet, there is no guarantee that the received packet’s content has not changed during the trip. So a developer has to add his own content checking in the implementation such as using checksums. The current model does not currently target this issue and defers it for future work.

- **Delivery guarantees:**
  
  Also UDP sockets provide no mechanism for guaranteeing the delivery of a packet. Such an issue can be dealt with using message receipt acknowledgements. In fact, the current model describes an Acknowledge message in the protocol formats described later for this purpose. However, the current implementation did not make
use of this message relying on the reliability of the network used and to reduce further programming overhead dealing with retries and acknowledgements.

- **Packet ordering:**
UDP sockets do not guarantee that multiple received packets are delivered in order, so a component that requires sending multiple ordered packets has to provide some way of an order identifier or tag to enable the client to reorder the packets after being received.

### 4.12.2 Problems with TCP Sockets:

- **Connection overhead:**
TCP sockets on the other hand show more reliability, delivery guarantees and in order delivery of packets. This comes on the expense of more overhead on establishing a connection. A component requiring a connection has to make a request and wait for the server to accept the connection request. After the connection has been made, a dedicated communication channel is open now between the two peers and if another component has to be targeted the whole cycle has to be repeated. This overhead has no impact on applications on which a dedicated connection between two peers is required especially if it is used to transfer large amounts of data like over the Internet. But, in situations where speed is an issue and multiple open connections with several components are required, this would introduce a tedious procedure.

- **One peer per each open socket:**
As mentioned before, TCP sockets allow only one peer per open socket and so it does not support broadcast messages. In many situations, when a message has to be received by more than one target, broadcast messages come in handy as in the cases with
multiple replicas or data synchronization between similar components. However, the uncontrolled use of broadcast messages introduces unnecessary network saturation. Fortunately, group multicast can serve better in this aspect and is supported by TCP connections. Group multicast serves to identify the recipients only of the concerned message so as not to overload the network with unnecessary delivery to all nodes. Moreover, group multicast is not implemented in several socket models for different programming languages on different operating systems, so the choice of platform and programming language, as well as the application requirements, has a great impact on the choice between group multicast and network broadcast topologies.

The proposed system depends on message passing in its inter-component communication and as will be shown, the system is better off using asynchronous message passing to allow for parallelism using the designed model which is not inherently supported using synchronous messaging as in RPC-based models describe before as Java RMI. In these models, parallelism can be achieved through implementing multithreaded components.

4.13 Features of The Model:

4.13.1 Registry:

The registry or the component database is represented by an XML file that registers all components that reside on a node in the system, together with the path to the component that implements it. Once a Broker has access to this file it can serve to get an object reference to the requesting client. It instantiates the component and gives it the reference to the client so as to communicate with it (Figure 10). A reference in this
context means an IP address mixed with a port number in the form “xxx.xxx.xxx.xxx:pppp”. A sample of the components database looks like:

```xml
<COMPONENTS>
  <COMPONENT NAME="CSolver"
    PATH="E:\AUC\cs599\Source\Windows\VC\scksvr\Debug\scksvr.exe" />
  <COMPONENT NAME="VBSolver"
    PATH="E:\AUC\cs599\Source\Windows\VB\VBSolver.exe" />
  <COMPONENT NAME="JSolver"
    PATH="E:\AUC\cs599\Source\Java\JSolver.bat" />
</COMPONENTS>
```

### 4.13.2 Interfaces:

Object interfaces are built in the proposed model using XML files that are accessible to Brokers to let clients use server components more efficiently. Each server component has its own interface repository that contains a description of the exposed methods, their arguments and their data types. A client may invoke a GetConvention on a Broker to get the result as an XML message describing the parameters needed and their types to formulate the RPC to the server. This feature is not a mandatory feature as it adds more complexity to the system, however, it adds to the flexibility and versioning capability mentioned before. For example to plug in a new version of an object, you just have to put in the new compiled object and its interface file in the Brokers repository without having to notify other objects or to stop the system.

### 4.13.3 Security:

Security is built upon XML access rights files that are implemented at two levels (Figure 10):
1) **Broker accessible files:**

These files are only accessible to the Broker and are used to authorize a client to instantiate a server component. The Broker looks up the necessary access rights for the requesting client and authorizes it accordingly or sends an error message otherwise. An example:

```xml
<Access>
  <Component Name="CSolver">
    <Sources>
      <Component Name="master"/>
      <Component Name="VBSolver"/>
      <Host Address="198.162.0.41"/>
      <Network Address="10.0.0.0"
        Mask="255.255.255.0"/>
    </Sources>
  </Component>
  <Component Name="Jsolver">
    .
    .
    .
  </Component>
</Access>
```

This example means that component *CSolver* can be accessed only by a component named *Master* or component named *VBSolver* or any component from a host address of 198.162.0.41 or any component on the C subclass network of 10.0.0.0.

2) **Server accessible file:**

The server accessible file provides more details to the security of the methods implemented by the server component itself. It provides components access to individual methods within the server depending on their IP address, port number or Component name. This would provide a cover for the security holes introduced by opening one port for each component that is instantiated instead of communicating through one port which is the runtime module in
conventional systems. Thus, a component may provide access rights to critical methods only to components connecting from a certain IP address and a certain port (for example an administrator’s PC).

4.13.4 Fault Tolerance and Load Balancing:

Fault tolerance is an issue that has been discussed in almost every distributed model implemented. There are several approaches to implement fault tolerance, but in the proposed model a simple method is to replicate readily required objects onto several nodes and use a broadcast mechanism over UDP sockets to get an object reference for a certain object. This way the client guarantees that he would get an object reference to the required server object if there were one on a running node. Also, if a node harboring the required server object is down, the other nodes will take over automatically. However, this technique would necessitate embedding an automatic shutdown mechanism for servers if they do not receive an initial request within a certain period of time, as implementing this broadcast method would leave the nodes cluttered with unused server objects. Also, this method allows for selecting the node with the quickest response to serve the required object making some kind of a load balancing. However, as this technique depends on other factors than node load, as network connection, node location, network subclasses, a more reliable technique for load balancing should be implemented using reference counts and reliable load indicators. Such a technique can be implemented using a central node with a specialized service component for registering node loads. Each server has the obligation to register with this service upon receiving the initial request and deregister upon shutdown.
4.13.5 Transparency:

Transparency in distributed systems can be any of several forms including access, location, concurrency, replication, failure, migration, performance and scaling transparencies [10]. The current system in its basic form implements access transparency using direct addressing mode or location transparency using broadcast addressing mode (see later). In comparison to the previously mentioned models, most of them implement access transparency only as the target node has to be determined prior to making the call. Also, the flexibility of the system allows changing of the platform, hardware architecture and even the programming language for better performance and scaling and thus achieving performance and scaling transparencies.

4.14 Summary:

In this chapter we overviewed the points of concern of existing distributed models and how XML has a positive impact on each of these points. We also reviewed the features and benefits of the proposed model and presented a global view of how the system components, network and communication channels and system files should look like. In the next chapter we are going to present a detailed system design.
CHAPTER 5
SYSTEM DESIGN

5.1 Basic Setup:

The basic setup of the system consists of a Broker component residing on each node that participates in the distributed system awaiting requests of client components that need to utilize the functionality of other components.

5.1.1 Component Registration:

Server components which are ready to expose their services are run once with the command line parameter “/r” to acknowledge the Broker residing on the same node of their existence and exposed interfaces. This is done through sending an XML message to the Broker on the port it is listening to. The Broker then stores the executable path and the component name in a special XML formatted file that resides in the same file system directory as the Broker executable that is of course made secure by the system administrator. It also creates an access file for the component and registers the access control list in that file. The access control list can be changed by starting the component with the “/a add <acl parameter>” command-line switch.

For example to add a host with an IP of 192.168.0.45 to the access list of the component, we would run the following command-line “component.exe /a add host 192.168.0.45” and to delete a component from the access list of another component we run the command-line “component.exe /a del component Jsolver”. The same happens on each node in the system and this concludes the basic setup.
5.1.2 Component Deregistration:

On the other hand, once a component is required to be out of the system, it is started with the command line parameter “/u”. This causes the component to send a message to the Broker on the same node, which in return removes the component’s information from the appropriate file and deletes the component’s access control list file.

5.2 Communication Scenarios:

Once the system is up, there are several scenarios that can occur:

5.2.1 Transparent Component Addressing (Broadcast Request):

Server components register themselves with different Brokers. If a client component needs to invoke a method on the previously registered server component which it doesn’t know its location. The GetObject method is send as a broadcast XML message containing the server component name to all Brokers. Each Broker component receives the message and searches its local files for the component name. The Broker that finds the component verifies the security access rights then starts the server component on the same node giving it the IP address and port of the requesting component. If up to this point the Broker fails to authenticate the requesting client or host, it sends an error message to the client component. If the client component times out waiting for a response, the component is considered not to be found.

The role of the Broker has now ended. The started server component now begins to communicate with the client component directly. It sends a result message containing the response to the GetObject message containing reference to itself in the form of “Address:port” which identifies the component on the system. The client then starts to invoke methods on the server component to utilize its functionality. It can use the server component as much as it wants and sends it a shutdown method call to shut it down after it has finished. Other
server components may have started on other nodes during the request. These server components wait for some predefined interval and do an automatic shutdown so as not to clutter nodes with unusable components (Figure 11).

5.2.2 Targeted Component Addressing:

Owing to the overhead broadcasts cause to the networking resources. Targeted component addressing would minimize this overhead by two ways:

a) Direct Broker Addressing:

In this mode, the client component happens to know the node address where the server component resides. It sends a direct message to the Broker on this node requesting an object reference for a server component. This could be incorporated with the previous scenario by adding the functionality to the client component to save the location of the node of a certain server component into local files for further reference. The Broker then authenticates the caller and starts the server component if everything goes well giving it the caller address and port on the command-line or sends an error message identifying the error that occurred to the requesting component. The rest of the scenario continues as in Transparent Addressing (Figure 12).

b) Direct Component Addressing (Broker Bypass):

In a distributed environment, a distributed file system may be present. A famous example is the Sun NFS distributed file system. If such an environment is present, it could be of great use to the shared file system implementation. The client component, this time, happens to know the file system path of the server component it requires to invoke. It then formulates the usual message in XML adding to it its location and port and starts the required server component passing it the message as a command line parameter and awaits the response. The server component, not knowing whether the Broker or the client component invoked it sends its IP address and port to the location on the command-line.
The client then uses the server component and then sends it a shutdown invocation to terminate (Figure 13).

5.2.3 Parallel processing:

The system proposed could be used as a message passing protocol for implementing parallel processing techniques. This scenario is based on redundant components that have the same functionality present on different nodes on the system and utilizes the scenario number (5.2.2.a) above.

The master component divides the problem to be parallelized into several smaller tasks. It sends a number of GetObject method calls equal to the number of processes it needs divided among the nodes in the system using Direct Broker Addressing. Each Broker, receiving an invocation from the master component, will spawn a server process passing it the source as a command line parameter. The master component then waits until it gets all object references it needs keeping them in an array or a linked list and then starts sending the same method invocation for the parallel problem to each of the server objects each with its own required parameters. The master component will start receiving the results from the invoked components and reassembles the parts of the main problem and solves it. Each of the invoked server components will await the shutdown message or the timeout to terminate and then die (Figure 14).
Figure (11) Transparent component addressing, 1) Broadcast message from component A, 2) Broker on node I found required component B and starts it, 3) Component B sends its address to component A, 4) Component A sends method invocation message on component B.

Figure (12) Targeted Component addressing 1) Direct message from component A to Broker, 2) Broker on node I found required component B and starts it, 3) Component B sends its address to component A, 4) Component A sends invocation message to component B.
Figure (13) Broker Bypass, 1) Component A spawned component B on an NFS, 2) Component B sends its address to component A, 3) Component A sends method invocation to component B.

Figure (14) Parallel processing, 1,2,3,4) Component A sent direct message to Broker on nodes I, II, III, IV, 5, 6, 7, 8) Brokers on appropriate nodes starts component B, 9, 10, 11, 12) Component B on each node send their addresses to the master component A, 13,14,15,16) The master component starts sending each B component its part of the problem to solve.
5.3 THE RPC PROTOCOL FORMATS

5.3.1 The <RPC> Tag:

All XML encoded RPC messages have a common tag that can be the root element of the XML structure of the protocol, the <RPC> tag. The RPC has only one attribute to identify which type of message is being sent. This attribute is the “TYPE” attribute. A “TYPE” attribute may have one of four values: “CALL”, “RESULT”, “ERROR” and “ACK”, each of which denotes a different type of message and the rest of the XML structure then follows differently according to the type of message.

5.3.2 The CALL message:

This is the actual RPC call message which is sent to the remote component to invoke a certain method on it. It is composed of 3 tags: The <RPC> tag, the <METHOD> tag and the <PARAMETER> tag.

5.3.4 The <METHOD> Tag:

The <METHOD> tag follows the <RPC> tag in an RPC Call message that is the “TYPE” attribute is set to “Call”. This tag has two attributes, the “TARGET” and the “NAME” attributes, and one child tag which can be repeated, the <PARAMETER> tag.

5.3.5 The <PARAMETER> Tag:

The <PARAMETER> tag identifies parameters to the RPC call concerned. It defines the type of the parameter and its value. This tag has only one attribute specifying the type of the parameter, e.g. “integer”, “string”, “float”, …Etc. These types depend on the required implementation. The tag contains the value of the parameter. An example of an RPC call would look like:

```xml
<RPC TYPE="Call">
  <METHOD TARGET="Utility" NAME="Sum">
</RPC>
```
5.3.6 The Result Message:

The RPC result message is sent by the invoked component in return to an RPC call message from a client component. The RPC result contains an `<RPC>` tag with a child `<RESULT>` tag. An example RPC Result message would look like:

```
<RPC TYPE="Result">
  <RESULT TYPE="integer">
    55
  </RESULT>
</RPC>
```

5.3.7 The `<RESULT>` Tag:

The `<RESULT>` tag identifies a result message. This tag has a “TYPE” attribute that identifies the type of the return result. The tag contains the result value. An example of a result message is shown below:

```
<RPC TYPE="Result">
  <RESULT TYPE="integer">
    55
  </RESULT>
</RPC>
```

5.3.8 The Error Message:

This type of message is sent in response to an error or exception that has occurred in the invoked component due to failure of validation of parameters or error in the result or execution of the invoked method. This error can be generated from the component itself or can be propagated across a series of chained component calls. Also a Broker can respond to a client with an error message if it does not find the required component. It contains an `<RPC>` tag with `TYPE = "ERROR"` and an `<ERROR>` tag.
5.3.9 The <ERROR> Tag:

The <ERROR> tag appears in an RPC message to notify that an error has occurred in the invoked component. The <ERROR> tag has one attribute that is the “NUMBER” attribute. It identifies the error that happened by an error number. The tag contains the error message.

An example RPC Error message would look like the following:

```xml
<RPC TYPE="Error">
  <ERROR NUMBER="1652">
    Type mismatch in Parameter 1
  </ERROR>
</RPC>
```

5.3.10 The Acknowledge Message:

This is the simplest type of a RPC message but not the least important. The message is sent to acknowledge a receipt of a call or a result so as to let the peer component retry sending if a packet failed. An Acknowledge message would contain an <RPC> tag with TYPE="ACK". The Acknowledge message looks like:

```xml
<RPC TYPE="ACK"/>
```

5.4 Summary:

In this chapter we presented a detailed scenario for component communication including the transparent, direct and parallel processing scenarios. We also described component registration and deregistration with the naming service represented in the Broker component. Finally, we explained the structure of the XML-RPC message formats with a detailed description of the tags used in these XML RPC messages and how to formulate them. A brief description of XML can be found in the appendix.
CHAPTER 6

IMPLEMENTATION AND BENCHMARKING

6.1 Introduction

Measuring the throughput of the proposed system had to be dependent on just the formulation and message communication overheads. The RPC performance is measured by the null RPC. It is defined as an RPC without parameters that executes a null procedure and returns no values. Typically, it carries system data and no user data and comprises around 100 bytes of data [10]. It was reported that the best time for a null RPC is about 1 millisecond on a 10 megabits per second network, and that the network transfer time for the same amount of data is about 0.1 millisecond. The difference is accounted for by the overhead caused by RPC-related user and operating system procedures. Also studies showed that as RPC delay is directly proportional to RPC data size until RPC size increases beyond a packet size that causes a jump in the RPC delay for each extra packet needed. RPC Delay can be accounted for by Marshalling, Data transfer, Packet initialization and Thread scheduling.

6.2 Methodology:

The chosen methodology for benchmarking our design was to implement a null XMLRPC which contains a virtual call from the client to the server and an acknowledgement from the server to the client. This experiment was done on two stages: the first is sending the GETOBJECT request to the Broker to instantiate the component and give it the client’s address and port. After the client is instantiated, it then sends the acknowledgement to the requesting client. This RPC time measure would
include time for instantiation as well as RPC time, so another call similar to the first one is made to the client directly after instantiation to measure null RPC time. The request method was as follows:

\[
\text{<RPC TYPE="CALL"} > \\
\text{<METHOD NAME="GETOBJECT" > } \\
\text{<PARAMETER TYPE="STRING" > } \\
\text{COMPONENTNAME} \\
\text{</PARAMETER>} \\
\text{</METHOD>} \\
\text{</RPC>} \\
\]

and the response message was as follows:

\[
\text{<RPC TYPE="ACK"/>} \\
\]

Accordingly the whole data to be transferred was around 120 bytes, which was within the required range for a null RPC.

6.3 Development Tools:

The development was done on an Intel architecture PC hosting Windows 2000 Professional equipped with 256 RAM, a PIII 750 MHz processor and a 30GB Hard disk.

6.3.1 Microsoft Visual Basic 6.0:

The Microsoft Visual Basic environment was used to build a Broker for listening to client’s requests, reading an XML files for the paths for components’ executables and instantiating them accordingly. It was also used to build the client process that sends the request to the Broker and waits for receiving a response from the server component and timing instantiation and null RPC times. Finally, a server component was also built using VB to be instantiated by the Broker. The client process could specify which component type and which addressing method to use through a simple user interface. Socket communication was done using
the built-in Microsoft Winsock control which is a wrapper of the basic API system calls and XML parsing was done using the Microsoft XML parser ActiveX Library available with the Windows 2000 professional distribution and can be downloaded at the Microsoft Developer Network site [36]. Both components are ActiveX components based on the COM+ infrastructure describe before.

6.3.2 Microsoft C++ 6.0:

The Microsoft Visual C++ environment was used to build a server component. Socket communication was done using the MFC CAsyncSocket class which is also a wrapper class of the system API and XML parsing was done using the Microsoft XML parser DLL, the same one used for VB. A native XMLRPC library was built into this component.

6.3.3 Java JDK 1.3:

The Java language was also used to build a server component using the Java Developer Kit (JDK) version 1.3 [47]. Socket communication was done using the built in DatagramSocket class in blocking mode, and XML parsing was done with the apache Xerces library downloadable at the Apache website [48]. The XMLRPC library was built in native java code and compiled into the component.

6.4 System Components:

6.4.1 The XMLRPC Library:

An XMLRPC library was built to expose a set of utility functions to simplify the process of encoding and decoding of XML messages, an essential task of each component in the system. Each function in the library simplifies the access of XML Documents and maps system related
functions to XML related tasks. Functions in the library are described as follows:

- **GetSource()**: A function that extracts the source address of the XML message sender embedded as an attribute to the main RPC tag in the form *ip-address:port*.

- **GetMethod()**: A function that extracts the name attribute of the method tag of the XML message call.

- **GetParameter**: A function that extracts a specified parameter by name or by index from Parameter tags of the method tag of the XML message call.

- **GetType**: A function that extracts the attribute Type of the RPC tag of the XML message.

- **GetPath**: A function that searches and extracts the path of a specific component on a node from the components database XML file given the component name. It is used by the Broker to fetch the components file for the requested component.

- **RegisterComponent**: A function that registers a component in the components database by building a component node in the XML document and adding path and name attributes to it. It is used by the Broker component to register and alter component information in the components database XML file or create a component database file if one does not exist.
- **UnregisterComponent:**
  This function is the reverse of the previous one. It removes a component’s node from the components XML document. It is used by the Broker component to remove a component entry from the components database file.

- **FormatCall:**
  This function encodes an XMLRPC call into an XML message to be sent as a request. It creates an RPC tag with Type Call, a method tag with its attributes and parameters given to this method with their type and value.

- **FormatResult:**
  This function encodes the result of a method invocation on a remote object into XML message to be sent to the caller component. It creates an RPC tag with Type Result and creates a result tag with its type and value.

- **FormatError:**
  This function encodes an error message as an XML message to be sent to the calling component. It creates an RPC with Type Error and creates an error tag with error number and error message attributes.

- **FormatAck:**
  A function that returns an encoded Acknowledge XML message by creating an RPC tag with Type ACK.

- **AddSource:**
  This function appends the source address of the sending component to the RPC tag of the XMLRPC message. It is used by the Broker to label each coming message with its source to be able to respond afterwards.
The library was implemented as a COM dynamic link library in Visual Basic to demonstrate interoperability of the proposed system and COM. As mentioned before the library was also ported to Visual C++ as a native C++ class and to Java as a native Java class.

6.4.2 The Broker:

The Broker component is the naming service component. It is the only component in the system that is always running on any node that is part of the system. The purpose of this component is to listen on a preset system-wide port that is the same on all nodes for incoming requests. Port 5000 was used in our setup but it can be changed according to implementation, availability and needs. The Broker receives incoming requests on this port in the form of XML formatted messages that encode the required operation from the component.

Basically the Broker is waiting for a *register*, *unregister* or a *getobject* methods but it can be extended to allow for more functionality as the system needs such as logging and shutting down components. If a message is received, the message is labeled with its source and queued into a FIFO queue. Also the implementation of the queue is left for the developer. In this case the Broker runs two threads of control (a multithreaded process), one for receiving and queuing requests and the second thread for actually handling the received requests.

The multithreaded nature of the Broker component was due to the unreliable nature of the UDP sockets. A small experiment done with a prototype of the Broker showed that without actually saving the requests and handling them directly would cause the Broker to lose about 20% of the requests, but with queuing and multithreading the Broker did not miss
any requests. This can be explained by the fact that saving the request together with its source did not represent much of an overload on the system. The implementation of the queue used is not restricted to a FIFO queue; it can be further extended to implement more complex and productive implementations. An example of such a variation is a priority queue. Implementing a Broker without a queue is possible in simple low load systems but it is not such a good idea.

The other thread of the Broker is busy handling requests from the queue one by one. It decodes the message and fetches the path for the executable of the required object from an XML-Formatted file called components.xml which resides on the same node and contains all the registered components within this node. This file is edited by the Broker upon receiving the register and unregister method calls from system components to register or unregister their selves from the system respectively. When it finds the path for the component, it executes it and passes it the address and port of the caller process (decoded also from the stored message). From now on the spawned component is all by itself and should do the communication with the requesting component directly. The Broker then returns to serve more requests.

Although Visual Basic does not support multithreading programmatically, using the DoEvents statement in the queue-handler function and the event driven socket implementation has provided a workaround for this. The process of registering and deregistering a component is merely adding a <COMPONENT> tag to the Components.xml file with the appropriate attributes (NAME and PATH) or updating the attributes for an already existing component and
removing the whole tag for a component to deregister it as mentioned in the function implementations in the XML library.

6.4.3 The Client:

The Client component was built only using Visual Basic, being the only component that needs to be unique on the system. Visual Basic was chosen for simplicity. The Client component was built with various options for testing: Direct and Broadcast addressing, the ability to set the group of hosts to be targeted, number of repetitions of invocations and specifying which type of components should be spawned, VB, VC or Java components. Each of these, was registered in the components database under a different name: VBSolver, JSolver and CSolver respectively (Figure 15).

The Client component starts by saving the system clock in a local variable and sending a request message for each of the targeted nodes to spawn a server component of the specified type. This method contains an encoded invocation of the GetObject method from the Broker. In case of a Broadcast protocol it sends one broadcast message to all Brokers. Each of the spawned processes then sends back the Client its socket address in the form of an encoded XML result message (address: port) so as to let the Client begin invoking methods on it. Upon receiving the results from the spawned components, the Client stores the time difference between the save time and the current time to be referred to as the instantiation time. The Client then stores the addresses of spawned components in a collection object and saves the system time once again. It starts sending each component an XML encoded message for the GetObject method and upon receiving the results it calculates the time difference again as before
to be referred to the null RPC time. After it has got all the results it needs, it sends a shutdown call to all spawned processes using the collection of object references it has and displays the results (Figure 15).

![Image](image1.png)

*Figure (15) The Client Component showing multiple options and the status window*

### 6.4.4 Server Components:

Server components are the ones that expose methods to provide functionality to the system. In our case, they provide just the functionality of sending their address and port or responding with an Acknowledgement, in addition to the *SHUTDOWN* method to end execution. Server components were built as single threaded components owing to the simplicity of the system (just one thread for receiving and dealing with requests). The server component accepts a socket address of the form address: port as a command line argument to identify its peer for starting communication. It also accepts a /r or a /u switch to register or
unregister itself with the Broker residing on the same node respectively. The component is spawned by the broker and immediately responds to the caller by sending it its address and port. It then gets an RPC call with the method GetResponse. The component then responds with an RPC Acknowledge message. It waits until it receives the SHUTDOWN from the caller and then ends execution. In case of failure to communicate the component waits for a preset time (in this case one minute) and shuts down itself if it does not receive any messages during this time.

The Java component was called using a batch file called JSolver.bat that initializes the Virtual Machine on the concerned node giving it the class path and the necessary parameters. The path of the batch file was inserted in the component database in association with the name ‘JSolver’ during the process of registering the component.

6.5 The Java RMI System:

A Java RMI system comparable to the above described system was built for the purpose of comparison. The system consisted of a client component, a server component and the RMI registry as the Broker. The interface RMISrvr was built to implement the Remote interface, it exposed only the getResponse() method. An RMISrvrImpl class was built extending the UnicastRemoteObject and implementing the RMISrvr interface. The main purpose for code of the getResponse() method was to return an acknowledgement to the client. The RMISrvrImpl class was compiled with the RMI compiler to produce the server stub and the client skeleton. The RMIRegistry was then run on the server node, and the RMISrvrImpl was run on the same node to register itself with the RMIRegistry using the name RMISrvr. The RMIClnr class was built to save the system time and then get a reference to the RMISrvr using the
Naming library. The RMIClntr would then invoke the getResponse() method on the RMISrvr reference and calculate the time difference to be referred to as the instantiation time. The RMIClntr would then save the system time and make another invocation of the getResponse() method and recalculate the time difference, this time referred to as the null RPC time.

**6.6 Benchmarking Setup:**

The PCs used for testing were 3 Pentium III based PCs ranging from 500 MHz to 800 MHz equipped with 128 RAM, a 3COM 100 MB/sec Ethernet network card and a 20 GB Hard disk drive hosting Windows 2000 operating system.

The benchmark was tested using 3 different setups:

?? **Direct addressing:** using direct messages to Brokers on server nodes.

?? **Broadcast addressing:** using Broadcast messages to all Brokers on the local network.

?? **Java RMI:** using the RMI addressing topology.

The Direct and the Broadcast addressing setups were run once with every component type (VBSolver, JSolver and CSolver) and results for 100 invocations were obtained for components on local node and remote nodes separately. Setting up the benchmark was done by running the broker on each node to be used in the system and then registering the server component by running it with the switch /r. The Client component is run from any node which may or may not have a running broker and starting the benchmark by different combinations as mentioned before.
In the case of Java RMI, the experiment was run with the RMIRegistry running on the local node once and another with the registry on a remote node.

6.7 Summary:

In this chapter, a detailed explanation of a prototype implementation of the proposed system was given including the ready-made libraries and components used and the libraries built specifically for the benchmark at hand. The XMLRPC library functions used to encode and decode XML-RPC messages were explained. Also a detailed explanation of the functionality of system components (Brokers, Servers and Clients) was given and the programming technique used in building them. Finally, the experimental details and the required setup were given to describe the environment in which the testing was done.
7.1 Introduction:

Experiments were run with different types of components VB, VC and JAVA using the proposed design. With each component the direct and broadcast addressing were used and a null RPC was tested for performance after instantiation. For each different run of the experiment the timing in milliseconds was recorded and the mean and standard deviation of 100 invocations were obtained. Also results obtained from remote nodes were compared to those obtained from the local node to verify the efficiency of the mechanism relative to local IPC mechanism. For the sake of comparison a similar procedure was done using the Java RMI mechanism. The results obtained can be seen in the following tables with times expressed as Mean±SD. It is worthwhile noting that timing accuracy depended on the system clock resolution.

<table>
<thead>
<tr>
<th>Type</th>
<th>Visual Basic</th>
<th>Visual C++</th>
<th>Java</th>
<th>RMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct</td>
<td>140±2 ms</td>
<td>147±6 ms</td>
<td>807±42 ms</td>
<td>285±7 ms</td>
</tr>
<tr>
<td>Broadcast</td>
<td>140±3 ms</td>
<td>163±6 ms</td>
<td>1113±1121 ms</td>
<td>N/A</td>
</tr>
<tr>
<td>Null RPC</td>
<td>6±4 ms</td>
<td>20±2 ms</td>
<td>413±6 ms</td>
<td>0 ms</td>
</tr>
</tbody>
</table>

*Table (3) Results in milliseconds of testing Direct instantiation, Broadcast instantiation and the null RPC with different components (Visual Basic, Visual C++, Java and Java RMI) on the same node*

<table>
<thead>
<tr>
<th>Type</th>
<th>Visual Basic</th>
<th>Visual C++</th>
<th>Java</th>
<th>RMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct</td>
<td>93±7 ms</td>
<td>104±16 ms</td>
<td>609±20 ms</td>
<td>282±6 ms</td>
</tr>
<tr>
<td>Broadcast</td>
<td>91±3 ms</td>
<td>92±4 ms</td>
<td>554±249 ms</td>
<td>N/A</td>
</tr>
<tr>
<td>Null RPC</td>
<td>1±4 ms</td>
<td>2±4 ms</td>
<td>411±15 ms</td>
<td>0 ms</td>
</tr>
</tbody>
</table>

*Table (4) Results in milliseconds of testing Direct instantiation, Broadcast instantiation and the null RPC with different components (Visual Basic, Visual C++, Java and Java RMI) on remote nodes*
7.2 Observations:

Using Direct and Broadcast addressing, the instantiation times for VB and VC components were comparable, with VB components being a little more responsive, while that for Java components was 5.5-7.5 times more than that of both VB and VC and also about 2.5 times that of RMI components (Figure 16 and 17).

The instantiation times for components residing on the same node were about 1.5 times those for components residing on remote nodes using the Direct Addressing method and reaching about 2 times using the Broadcast Addressing method for all types of components (Figure 16 and 17).

Figure (16) Chart showing instantiation times in milliseconds using Direct Addressing for the VB, VC, JAVA and RMI components
Instantiation times for components residing on the same node using Broadcast Addressing was identical to that using Direct Addressing for VB components, while for VC components it was a little higher and for Java components it reached about 1.4 times. However, for components on remote nodes, the instantiation time using Direct Addressing was almost identical for VB components and about 1.1 times higher than that using Broadcast Addressing for VC and Java components (Figures 18 and 19).
As for the null RPC results, VB and VC showed values comparable to those for RMI components that actually were beyond the time granularity for the PC clocks for both local and remote components. For
Results

VB and VC local components, the value was a bit higher reaching from 6 (for VB) up to 10 times (for VC) that of remote components. The Java components showed high variations in null RPC times for local and remote components which were much higher and incomparable to those for VB, VC and RMI components. However, these values were nearly identical on local and remote nodes (Figure 20).

![Figure (20) Chart showing null RPC times in milliseconds for the VB, VC, JAVA and JAVA RMI components](image)

**7.3 Analysis:**

The instantiation of VB and VC components comprises minimal overhead on a Windows platform, as they are native operating system executable components. However, the Java and Java RMI components rely on the Java Virtual Machine (JVM) for instantiation, which can be the reason for the lag in instantiation time. Moreover, the Java RMI components are instantiated through an already running JVM and RMI registry, hence the better performance than Java components using the proposed methodology which are actually standalone Java applications.
This outlines that as though Java RMI is the simplest and most efficient method for distributed Java computing, yet instantiation carries some overhead. However, using the proposed methodology for native operating system executables can have a much better response time with fewer overheads.

The local components instantiation time was always lagging behind that of remote components which may be explained by the fact that the local machine carries the client process and a Broker for instantiating server components which might present an added overhead, while the remote node only has a Broker running and is otherwise doing nothing. This fact becomes more obvious with Broadcast addressing were the Broadcast messages saturate the networking layer adding for more delay. However, instantiation and response of remote components with Broadcast Addressing was far better than with local components. The fact that, the proposed mechanism may not be the best alternative for local IPC is still under evaluation and other IPC mechanisms should be tried for local components.

Broadcast Addressing produced better results for remote components. This may be due to the fact that a Broadcast message is not sent to a definite address and that the socket used is not bound to a certain peer that has to be changed when trying to send to another peer as in the case with Direct Addressing. This allows for releasing such a message to all nodes on the network simultaneously and allowing nodes to respond more promptly but carrying the overhead of network saturation. An alternative to be studied is the group multicast algorithm where applicable.
Null RPC times for remote components showed promising results for VB and VC that approached nullity as RMI components. Considering the value of 1 millisecond as the best null RPC time on a 10 MB network, and that on such a network the transfer time accounts for 0.1 millisecond, we can deduce that 0.9 milliseconds are taken by other overheads of the RPC. Therefore, on a 100 MB network things are not much different, as most of the null RPC time is spent formulating the message and making other operating system calls to execute the RPC. The Java components still showed significantly higher values than VB and VC although the JVM is up and running. This may be accounted for by the socket implementation differences between Java and the Windows API and by the fact that messages are crossing the Windows/JVM boundary to reach Java components causing more overheads than components passing messages using the same API as VB and VC. It is important to remember that the client component which was responsible for the timing processes was built with VB. A further study has to be done by building a client in Java and verifying this conclusion.

7.4 Summary:

In this chapter we presented the results of the test case we provided and the comparison between different results obtained from different component implementations. We also compared the functionality and performance of Visual Basic, Visual C++ and Java components in terms of instantiation and RPC times while we also demonstrated interoperability with COM and RMI technologies. Finally we commented on the obtained results and provided explanations wherever possible.
CHAPTER 8

CONCLUSION, OPEN ISSUES AND FUTURE WORK

8.1 Conclusion:

The proposed model and technologies used provide a solid foundation for the design of a modern distributed object model. The simplicity and flexibility of the proposed technologies favor their use in developing more standardized, extensible and interoperable models. However, several enhancements and specifications have to be outlined to make the proposed system more suitable as a standard for distributed object model design. The ease of use of XML messages has proved to incur minimal overhead on the system functionality while providing the utmost flexibility for implementing several issues as security and name service transparency. Although sockets are a little tedious to program, most (if not all) of the modern programming languages provide wrappers, libraries and components to facilitate socket communication. While UDP sockets are less reliable than their TCP counterparts, they provide less overhead and more flexibility as a communication method, but more work should be done regarding inherent problems like delivery verification, ordering of packets and large sized packets where TCP sockets become more of use. These problems are to be taken care of as a part of the system implementation and according to the developer’s requirements. Finally, the proposed method has preserved cross boundaries interoperability as seen by the implementation, which is a crucial part in designing any new model for distributed computing.
8.2 Deliverables:

One of the most important activities to help implement such a system is to formulate a set of XML Schemas for the previously mentioned messaging protocols. In spite of the fact that they are simple, more complex systems will need their components to verify such messages against a well-known Schema and reject those that do not conform to the rules. Also a developer should have a set of references to the structure of a message call before trying to implement system components.

As mentioned before, certain components have to expose basic interfaces that are mandatory to the system as the Register and Deregister methods of the Broker. Such interfaces should be well documented and clearly explained with all their parameters to ease the process of development and enhancement of such a system.

Last but not least, although such a system is considered simple, yet the delivery of wrapper libraries that mask the necessity of threading, XML encoding/decoding and low level socket calls might be a life saver to the less experienced developer involved in the development of more complex systems with more functionality.

8.3 Open Issues:

8.3.1 Security:

As a new and emerging concept, security should always be reconsidered after proof of concept. There are two open issues as concerning security:

Firstly, XML messages being entirely in text form are more liable to sniffs and hackers and easier to understand. Incorporating online
encryption as mentioned before and adding this capability to the wrapper library for formulating the encoded RPC introduces a solution to this issue.

Sockets are an easy way for a network hacker to spy on, especially with the development of more complex and smart network sniffing programs. May be communication over a secure sockets layer would be safer for those with critical data. Furthermore, The policy of opening ports on a system is always introducing security holes in the system. However, the system implementation as previously explained allows for ports to be open just for the time of communication and also, opened ports are not previously known as a component uses the first available port on the system. Moreover, the implementation of server side security would add more to this approach as certain privileged methods which have access to critical system resources may be only granted to components on certain nodes or even certain ports.

8.3.2 Replication:

All distributed systems should provide a way or another for performance increase, load balancing and fault tolerance. This means replication of components. The proposed system allows for replication components on different nodes as shown in the parallel processing example. But, now comes the question of how will a Broker with a least load acknowledge other Brokers that it will serve the client’s request and will it be able to stop them from spawning the desired components on all nodes with the requested server component. Another way of doing it, as mentioned before, is the reference count where a certain system component is acknowledge when a component is being served to a client and registers its reference. Once a client requests a server component, it should invoke a GetLeastLoadNode on this special service component to
directly address the broker on this node to serve the component. This way adds some more work for a client but may be there can be some way to incorporate this functionality in a Broker to relieve the client of such a round trip.

8.3.3 Large Message Packets:

Using UDP packets is a very simple and lightweight approach and can be very relieving in most circumstances, however, when it comes to large amounts of data such as object serialization. There are times also when you want to send a query for example as a parameter and get a record set that contains data as a result. In this case, the resulting data may be above the capacity of the UDP packet and there has to be some implementation that handles large packets and transfer them in chunks between the components. The developer would either use TCP sockets and establish a dedicated connection to transfer such data, or stick to UDP sockets and handle transfers over multiple messages. However, using UDP sockets, issues of reliability, deliver guarantees and packet ordering has to be dealt with in the implementation as described later.

8.3.4 Compatibility and Intercommunication:

There is no way one can replace a long implemented and stable system. And the best way to evolve is to try to communicate with it. Building the capability to translate RPC requests to and from CORBA, RMI and COM requests would be more like bridging the gap. This can be implemented into the Broker, the system components or just certain components in the system that expose this functionality and can be used by other components to communicate with the outside world. The way to implement this is still under research and has to be tested and proved feasible.
8.3.5 Internet Scalability:

Nowadays, the whole world has transferred to a large network, the Internet. Any emerging system should provide a way to communicate with objects or services on the web. Microsoft has gone a great way towards this through the SOAP project. SOAP is using the universal language XML that allows communicating with other systems on the Internet that use SOAP or XML. Also adopting the system on an Internet or WAN basis is not a tedious process but may provide an obstacle using UDPs for such RPCs. There must be a way to implement more reliably delivered messages to the servers and to the clients over unreliable communications as phone lines. Also Microsoft SOAP standards do not up till now include a well-defined security framework which means that a developer has to implement security for his components by himself.

8.3.6 More Languages and Platforms:

Implementing the proposed model by using other languages as Perl, PHP, Ada or Python has to be tested. Also using other platforms as Linux, Unix and the Mac OS seems feasible in the proposed model’s context but it has to go a long way to find out the obstacles, pit falls, incompatibility issues and certain precautions to be taken using XML and Sockets. Although parsers for XML exist for almost any existing language, the Socket interface can be tedious to use or not implemented. Some research has to be put into work to verify these issues.

8.3.7 UDP problems:

As described before, the current model does not mention how to target problems with UDP sockets. Further work has to be done in wrapper libraries to solve these issues. One of the most important is the issue of delivery guarantees which has to be dealt with using a mechanism of receiving an acknowledgement when a message is sent to
confirm receipt and also to add a message checksum to confirm message content. The developer then can implement a retry loop or timeouts when no acknowledgement is received and the topology of implementation is left to be decided according to the system’s requirements. Other issues like packet ordering are more important in multimedia applications and can be dealt with optionally according to the functionality required.

8.4 Summary:

In this conclusive chapter we provided some future work and open issues that are still to be studied as the proposed system is concerned. We suggested a set of deliverables to be handed in with the system specification to aid for further implementation and development. We also pointed out points of weakness and criticism in the proposed system with possible solutions to these points that need further study and confirmation.
APPENDIX

INTRODUCTION TO XML

What is XML?

Extensible Markup Language (XML) is a meta-markup language that provides a format for describing structured data. It emerged in an effort to increase the flexibility of HTML and decrease the complexity of SGML. It was developed by a group of SGML experts in 1996 in an attempt to select a subset of SGML to be well adapted to web applications [19]. In 1997, the W3 consortium adopted the idea and began some formal standardization of the idea. The goal was to standardize the method of transport, format and structure of web documents and to make the representation of various data formats in one type of document possible and unified to web users. This introduced new and novel ideas that were the result of continuous efforts to develop a more complex web-based environment. Some of the most prominent applications were well-structured documents as in libraries and museums, musical notation, math based language for documenting complex formulae and equations and genetic information representation in the medical field.

Why Use XML?

Although visual and user interface standards are a necessary layer for web applications, they are insufficient for representing and managing data [39]. Nowadays, the Internet provides an access resource to text and pictures. There are no standards for intelligent search, data exchange, adaptive presentation, and personalization. XML will provide a standard for data representation that will expand the Internet in much the same way that the HTML did a few years ago. The data standard will be the
vehicle for business transactions, publication of personal preference profiles, automated collaboration, and database sharing. Medical histories, pharmaceutical research data, semiconductor part sheets, and purchase orders will all be written in this format. It will open up a wide variety of new uses, all based on a standard representation for moving structured data around the Web as easily as we move HTML pages today. The data standard is XML and XML extensions.

**XML Syntax:**

XML defines the syntax for describing data. An XML document is considered well formed if it contains exactly one root element (the document element), and all the child elements are nested properly within each other. This means that both the begin and end tags of a given element should exist within the body of the same parent element. The following is an example of a well-formed XML document (components.xml):

```
<?xml version="1.0"?>
<hamburgers>
  <hamburger lowfat="dream on">
    <name>CowBurger</name>
    <description>Greasy and good.</description>
    <price>2.99</price>
  </hamburger>
</hamburgers>
```

HTML is essentially a specific case of an XML language with predefined elements and behavior [24]. These elements and their associated behaviors define how a given document will look like in a
Introduction to XML

Web browser and how it is used by the end user. In the same way that HTML provides a universal method to create user interfaces, XML offers a universal way to describe and work with data. XML allows developers to create their own XML vocabularies that are suitable for describing their particular data structures. A developer who utilizes XML need not worry about platform, operating system, language, or data store incompatibilities when interoperating with other systems.

**XML Namespaces:**

Because XML is truly about interoperability and everyone is free to create their own XML vocabularies, a serious problem arises quickly if different developers chose identical element names to represent conceptually distinct entities. To avoid these potential conflicts, the W3C introduced namespaces into the XML language. Developers use XML namespaces to provide a context for their XML document elements. XML namespaces allow developers to resolve elements to a particular implementation semantic. The following example illustrates how namespaces can help resolve any potential ambiguity:

```xml
<?xml version="1.0"?>
<hamburgers
    xmlns:purchase="http://fastfood.org/franchise/prices"
    xmlns:sales="http://fastfood.org/customer/prices"
>
    <hamburger lowfat="dream on">
        <name>CowBurger</name>
        <description>Greasy and good.</description>
        <purchase:price>0.99</price>
</hamburger>
</hamburgers>
```
One of the main advantages of utilizing XML as the universal standard for describing data is that any XML processor should give us the functionality we need to accomplish this goal. Developers should rarely (if ever) need to write their own XML processors [24]. Moreover, developers should be able to make use of the best processor on the market for their particular requirements without incompatibility issues. A standard XML processor can programmatically read any XML document and access any element name, body, or attribute. Even if we produced the XML document on a Windows-based system, we could easily send it to a mainframe system and use the mainframe's XML processor to interact with the same data.

**XML Validation:**

There should be some way to make sure that a particular class of XML document adheres to a certain format. A schema is basically a set of predefined rules that describe a given class of XML document. A schema defines the elements that can appear within a given XML document, along with the attributes that can be associated with a given element [24]. It also defines structural information about the XML document, such as which elements are child elements of others, the sequence in which the child elements can appear, and the number of child elements. It can define whether an element is empty or can include text as well as default values for attributes.
Document Type Definitions (DTDs) and XML-Data are both examples of specifications that outline how to describe XML document schemas.

**Document Type Definitions:**

The DTD language was invented specifically for defining validation rules for SGML documents. Since XML is a simplified subset of SGML, DTDs can also be used to define XML validation rules. An XML processor can use the DTD at run time to validate a given XML file against a predefined XML schema. The DTD syntax can sometimes be a bit complex. DTDs use different descriptive elements and different syntax from XML documents. An example is exclamation points, parenthesis, asterisks, angle brackets, and many others. DTDs also describe the relationship between elements and how attributes relate to different elements. Below is the DTD (hamburger.dtd) for the previously listed hamburger.xml file:

```xml
<!ELEMENT hamburgers (hamburger)*)
<!ELEMENT hamburger (name, description, price)>
<!ATTLIST hamburger lowfat CDATA #IMPLIED>
<!ELEMENT name (#PCDATA)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT price (#PCDATA)>
```

DTD syntax is not valid XML, therefore, XML processors must support the DTD syntax for describing schemas along with the XML syntax for reading documents. If we described schemas using XML, however, XML document validation would be much
easier to deal with for developers and, especially, XML tool vendors. The W3C is currently considering several alternate specifications that will alleviate the shortcomings of DTDs and provide enhancements to the grammar definition process.

**XML-Data:**

XML-Data is an alternative to the more complex DTDs proposed by Microsoft. XML-Data schemas are also referred to as XML schemas [24]. XML-Data schemas are well-formed XML documents contrary to DTDs which have their own syntax and structure. XML-Schemas are based on the same rules as XML documents and are valid XML documents. Owing to this fact, any tool used to work with XML documents can also be used to work with XML-Data schema definitions.

The following XML-Data schema produces a similar schema to the one defined above by hamburger.dtd:

```xml
<?xml version="1.0"?>
<Schema xmlns="schemas-microsoft-com:xml-data">
  <ElementType name="name" />
  <ElementType name="description" />  
  <ElementType name="price" />
  <AttributeType name="lowfat" />
  <ElementType name="hamburger" />
  <element type="name" maxOccurs="1" />
  <element type="description" maxOccurs="1" />
  <element type="price" maxOccurs="1" />
  <attribute type="lowfat" maxOccurs="1" />
</Schema>
```
XML-Data schema uses `<ElementType>` and `<AttributeType>` elements to define elements and attributes, respectively. These two tags define the structure and type of the element or attribute. Moreover `<element>` and `<attribute>` tags define an instance of an element or an attribute. Also the minOccurs and maxOccurs attributes define how many occurrences of a given element are allowed. The schema XML structure defines where the element is allowed to exist within the XML document.

**Processor (API) Technologies:**

As mentioned before, to benefit from XML, we must be able to programmatically access the data. A software module capable of reading XML documents and providing access to their content and structure is referred to as an XML processor/parser or an XML API.

Although developers are free to implement their own XML APIs, it is best to use industry-accepted standard APIs. By accepting an industry standard API, a developer can write code for a given API implementation that should be capable of running under any other compliant implementation of the same API without modifications.

There are two main API specifications that have gained popularity among developers today and are striving to become industry standards:
the Document Object Model (DOM) and the Simple API for XML (SAX).

DOM:

The Document Object Model is a defined standard for programmatically accessing the structure and data contained in an XML document. The W3C has approved the DOM Level 1 specification as a recommendation. The DOM is based on an in-memory tree representation of the XML document. When an XML file is loaded into the processor, it must build an in-memory tree that correctly represents the document. The DOM also defines the programmatic interface (including the names of the methods and properties) that should be used to programmatically manipulate an XML tree and access the elements, values, and attributes.

SAX:

One of the major downsides to the DOM standard is the overhead involved in loading the entire XML document into memory. For very large data files this can become tedious [24]. If large amounts of XML data are to be transmitted around the network or Internet, waiting for the entire file to finish transmitting before processing the file can be unacceptable. XML developers devised an alternate specification called SAX for this reason. SAX is a very simple XML API that allows developers to take advantage of event-driven XML parsing. Unlike the DOM specification, SAX doesn't require the entire XML file to be loaded into memory. As soon as the XML processor finishes reading an XML element, it calls into one of the custom event handlers to just-in-time process the element and its associated data. While this can greatly improve performance, developers do lose a degree of flexibility.
Introduction to XML

Transformation Technologies:

After we start using the standard DOM API to interact with XML data, it becomes quite tedious to extract specific pieces of data from large documents or to represent certain parts of an XML document as another format (such as HTML). We must manually write the code to scan the entire tree looking for specific elements that are required. These tasks were standardized by the W3C by introducing a specification for XML transformations called the Extensible Stylesheet Language (XSL) and a simple query language referred to as XSL Patterns.

XSL Patterns:

A pattern is a string, which selects a set of nodes in an XML tree. The selection is relative to the current node that the pattern is applied to. The simplest pattern is an element name; it selects all the child elements of the current node with that element name. For example, the pattern hamburger selects all the hamburger child elements of the current node. The pattern syntax allows us to identify the context of where a given element lives within a document.

XSL:

XSL Patterns help identify certain nodes within a given XML document, but it's still up to the developer to do something interesting with those selected nodes. XSL simplifies the process of transforming nodes from an XML format into another format. The need for this originated on the Web as developers wanted to take their XML data and
transform it into HTML for the user to view. XSL is also very useful for defining transformations from a given XML format to another distinct XML format. This makes interoperability much more feasible. With the simplicity of XML, developers no longer have to agree on a universal vocabulary for describing a certain type of data.

**Benefits of XML for the Internet:**

**Meaningful Searches:**

As data can be uniquely tagged with XML, it could be easily categorized in a standard way allowing agents to search these identified elements in a consistent way. Without XML, it is necessary for the searching application to understand the schema of each database, which describes how it is built. This is virtually impossible because every database describes its data differently.

**Development of Flexible Web Applications:**

Once data has been found, XML can be delivered to other applications, objects, or middle-tier servers for further processing. Also, it can be delivered to the desktop for viewing in a browser. XML, together with HTML for display, scripting for logic, and a common object model for interacting with the data and display, provides the technologies needed for flexible three-tier Web application development.

**Data integration:**

Searching multiple, incompatible databases is virtually impossible. XML enables structured data from different sources to be easily combined. Software applications can be used to integrate data on a middle-tier server from back-end databases and other applications as
email servers for example. This data can then be delivered to clients or other servers for further aggregation, processing, and distribution.

**Handling Data from multiple applications:**

The extensibility and flexibility of XML allow it to describe data contained in a wide variety of heterogeneous applications. Since XML-based data is self-describing, data can be exchanged and processed without having a built-in description of the incoming data.

**Local computation and manipulation:**

After being delivered to the client, data in XML format can be parsed and locally edited and manipulated, with computations performed by client applications. Data can be manipulated in various ways, rather than being merely presented on a web browser or a reporting tool. The XML Document Object Model (DOM) also allows data to be manipulated with several programming languages and data computations can be performed without additional return trips to the server. Separating the user interface from the data itself allows powerful applications, formerly found only on high-end databases, to be developed for the Web using a simple, flexible, open format.

**Multiple views of data**

Once data has been delivered to the desktop, it can be viewed in different ways. By describing structured data in a simple extensible manner, XML complements HTML, which is widely used to describe user interfaces. While HTML describes the appearance of data, XML describes data itself so having this data defined in XML allows different views to be specified, resulting in data being presented appropriately. Local data can be presented dynamically in a client specified manner or
user preference. CSS and XSL provide declarative mechanisms for describing a particular view of the data.

Partial updates

Data can be partially updated with XML, eliminating the need to resend an entire structured data set each time part of the data changes. Only the changed element is sent from the server to the client, and the changed data can be displayed without refreshing the entire user interface. On the contrary, with HTML, an entire page must be reconstructed if one item of data changes, even when the view is the same.

Delivery of Data on the Web

Because XML is an open text-based format, it can be delivered using HTTP in the same way that HTML can today without any changes to existing networks.

Scalability

XML completely separates the notion of markup from its intended display so developers can insert procedural descriptions of how to produce different data views in structured data. This is an incredibly powerful mechanism for migrating as much user interaction as possible to the client computer, while reducing server traffic and browser response times. In addition, XML allows for updates of individual pieces of data with only an update notice, greatly enhancing server scalability as a result of a far lower workload.

Compression

XML compresses extremely well due to the repetitive nature of the tags used to describe data structure. The need to compress XML data will be application-dependent and largely a function of the amount of data
being moved between server and client. XML can use the compression standard in HTTP 1.1 servers and clients.

**A Minimal Component Standard**

XML defines a minimal wire representation for data and message interchange [3]. This is the bare minimum of standardization needed to ensure that components can communicate. The core XML specification is extremely simple, as it only defines the syntactic ground rules for forming valid XML messages. While the World Wide Web Consortium (W3C) is rapidly developing additional standards on top of XML (for example, XLink and XML Schemas), the base-XML syntax has been fairly stable. The base XML syntax has proven to be quite flexible and adaptable to many applications, and despite its hierarchical nature, XML lends itself reasonably well to non-hierarchical data types.

**Platform, Language, and Vendor Independence**

Despite the hopes of platform vendors, the computing world will always be comprised of different programming languages, operating systems, and computing hardware. Being only a wire representation, XML is not coupled with one operating system, programming language, or hardware architecture. If two systems can exchange XML messages, they can potentially communicate regardless of their differences. XML does not necessitate an API or in-memory representation; therefore, it is fairly simple to host XML in an application. Several XML parsers are freely available for most programming languages and although there are several standardized programmatic interfaces for parsing XML, there is
no restriction on a specific API to use in order to interoperate with other XML-based systems.

**Accessibility:**

XML is very easy to understand, to read and consequently to author. This accessibility shared in XML's rapid acceptance [3]. XML messages can be easily created using a simple text editor or scripting language which is not the case with binary-wire protocols like DCOM, CORBA, or Java/RMI. While, many XML parsers provide facilities for generating well-formed XML, it is also possible generate XML using standard string manipulation facilities in any programming language. The simple text-based nature of XML is easier for debugging and monitoring distributed applications, owing to the fact that all component-to-component messages are easily interpreted and spied upon using a network monitoring tool.

**Extensibility:**

A system that is not extensible does not deserve to survive [3]. XML namespaces provide the Uniform Resource Identifier (URI) namespace to allow vendors to add attributes and elements to an existing XML vocabulary. However, the receiver of the message may have been developed independently from the sending application and this may introduce several potential problems. Depending on the interpretation of the receiver, the receiver may ignore unrecognized attributes and continue processing or may stop processing and signal a failure of request due to parsing error. Newer XML description technologies (such as Microsoft XML Data) allow XML vocabularies to be defined as either open or closed. Closed vocabularies cannot be extended beyond what is described
in the vocabulary schema while open vocabularies can. The receiving application has the role to determine how to interpret extended elements and attributes. Depending on the application, unrecognized extensions to a vocabulary can often be ignored but no failures due to parsing errors occur.

**Strong Typing:**

The use of open vocabularies and namespaces has enabled XML to support weakly typed communications. Although strong typing has many benefits (and is supported by XML using DTDs or their equivalents), it is extremely easy to build weakly typed systems using XML. This way XML can be extremely adaptable to generic application frameworks, data-driven applications, and rapid development scenarios with different requirements as regards typing.

**Interoperability:**

XML as a component integration technology does not completely solve the interoperability problem [3]. It only serves to move one step towards the solution, as different organizations are likely to use different XML vocabularies to represent the same piece of information. There are currently industry-wide trials to standardize domain-specific XML vocabularies; however, it is not known how far any of these efforts will achieve success in a specific domain.

Fortunately, the lack of standardized vocabularies can be solved using XML technology. Two competing vocabularies can be transformed into one another by application-level gateways and even a more promising solution lies in XSLT which allow one XML vocabulary to be
transformed into another by specifying the transformation rules. XSLT was originally devised to map XML to HTML, but has gained popularity in a variety of much more interesting scenarios.
REFERENCES AND WEB RESOURCES

1) Bettati R.: Remote Procedure Call (RPC)


3) Box D.: Lessons from the Component Wars: An XML Manifesto, DevelopMentor, September 1999


16) Hightower R: The Developer’s Guide To Understanding EJB 2.0, Trivera Technologies 2002

17) Hodes T and Katz R: Enabling XML Media Types, Computer Science Division, University of California, Berkeley, CA 94720-1776, 1998


24) **Randell B. and Skonnard A.:** A Guide to XML and Its Technologies, *DevelopMentor 1999*

25) **Rogerson D:** Components. *In Inside COM. Ed Microsoft Press Pubs 1997.*

26) **Roman E. and Oeberg R.:** The Technical Benefits of EJB and J2EE Technologies over COM+ and Windows DNA, *Sun Microsystems, December 1999*

27) **Rosenberger J:** Understanding the CORBA Architecture in *Teach yourself CORBA in 14 Days (Ziff-Davis Press) 1998.*


29) **Skonnard A:** SOAP: The Simple Object Access Protocol, *MIND January 2000*


31) **Tigue J. and Lavinder J.:** WebBroker, *XML for Distributed Computing: WebBroker, 1998*

32) **Tigue J.:** XML Enabled Mechanisms for Distributed Computing on the Web, *W3C Note 22-Sep-1997*

**Resources:**

33) CORBA home page:  
  [http://www.corba.org](http://www.corba.org)

34) OSF DCE home page:  

35) XDR Data Representation:  

36) Microsoft XML Parser:
http://download.microsoft.com/download/xml/SP/40SP1/WIN98M eXP/EN-US/msxml.msi

37) The apache XML web page:
http://xml.apache.org

38) The apache xerces xml parser for java:
http://gump.covalent.net/jars/latest/xml-xerces2/xercesImpl.jar

39) The Microsoft Developer Network:
http://msdn.microsoft.com

40) Trivera Technologies:
http://www.triveratech.com

41) Enterprise java beans home page:
http://java.sun.com/products/ejb

42) Enterprise Java Beans Documentation:
http://java.sun.com/products/ejb/docs.html

43) WebBroker Site:
http://www.datachannel.com/developers/webbroker/

44) Sun’s Java Home Page:
http://java.sun.com/xml/

45) Frontier web server
http://www.userland.com

46) The XML Schema specification
http://www.w3.org/TR/xmlschema-2/

47) The Sun’s JDK Home Page
http://java.sun.com/j2se/

48) The Apache XML page
http://www.apache.org/xml/