A Comparison of COM+ and EJB for Distribution in a Multitier Environment

Mats Sandin

Information Technology
Computing Science Department
Uppsala University
Box 311
S - 751 05 Uppsala
Sweden

This work has been carried out at
NTier Solutions AB
Östra Ängatan 31
S - 753 22 Uppsala
Sweden

Abstract

From the evolution of monolithic systems with no reuse, through client/server models with fat clients and data models reused to multitier systems with thin clients, both data model and business logic being reused, new technologies entered the market. Complex multitier applications are becoming more and more commonplace as companies rush to meet their information processing needs. Distributed object computing is a key enabler for these applications: It combines object-oriented concepts with client/server technology to produce a framework for building modular and scalable distributed applications at a relatively high level of abstraction. There are several techniques today used for distribution of data over networks. DCOM, COM+, EJB, and CORBA are some of these. This report is a comparison of these techniques, including COM though it is not a technique for distribution of objects, and will focus on different solutions to common problems. The emphasis of the report will be on two of the mentioned technologies.

Supervisor: Emil Gustafsson, NTier Solutions
Examiner: Roland Bol, CSD University of Uppsala
Passed:
1 Contents

1 CONTENTS .................................................................................................................................2

2 PREFACE .....................................................................................................................................5

3 BACKGROUND ..........................................................................................................................7

4 PRE STUDY OF TECHNOLOGIES ..............................................................................................9

4.1 COM ........................................................................................................................................9
  4.1.1 Overview ................................................................................................................................9
  4.1.2 Interfaces ..............................................................................................................................9
    4.1.2.1 Unknown .......................................................................................................................10
  4.1.3 Binary Interoperability ........................................................................................................10
  4.1.4 Globally Unique Identifiers (GUIDs) ..................................................................................11
  4.1.5 Packaging Transparency .....................................................................................................11
  4.1.6 Local/Remote Transparency ...............................................................................................11

4.2 DCOM .....................................................................................................................................13
  4.2.1 Overview .............................................................................................................................13
  4.2.2 Location and Packaging Transparency ..............................................................................14
  4.2.3 Free-Threading Model .........................................................................................................15
  4.2.4 Security ................................................................................................................................15
  4.2.5 Reference Counting and Pinging .......................................................................................15
  4.2.6 Load Balancing ..................................................................................................................16
    4.2.6.1 Static Load Balancing .................................................................................................16
    4.2.6.2 Dynamic Load Balancing ............................................................................................16

4.3 COM+ ..................................................................................................................................17
  4.3.1 Overview ............................................................................................................................17
  4.3.2 COM+ Catalog ....................................................................................................................17
  4.3.3 Load Balancing ..................................................................................................................18
  4.3.4 COM+ In-Memory Database – IMDB ...............................................................................19
  4.3.5 Queued Components .........................................................................................................19

4.4 EJB ..........................................................................................................................................21
  4.4.1 Overview ............................................................................................................................21
  4.4.2 Two Types of Beans ..........................................................................................................21
    4.4.2.1 Entity Bean ..................................................................................................................21
    4.4.2.2 Session Bean ...............................................................................................................22
  4.4.3 The Basic Part of an EJB System .......................................................................................22
    4.4.3.1 EJB Component ..........................................................................................................23
    4.4.3.2 EJB Container ..............................................................................................................23
    4.4.3.3 EJB Object ..................................................................................................................23

4.5 CORBA ..................................................................................................................................24
  4.5.1 Overview ............................................................................................................................24
  4.5.2 Definitions ..........................................................................................................................25
    4.5.2.1 Object ..........................................................................................................................25
    4.5.2.2 Servant ........................................................................................................................25
<table>
<thead>
<tr>
<th>Title</th>
<th>Changed</th>
<th>Ref</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A Comparison of COM+ and EJB for Distribution in a Multitier Environment</td>
<td>2001-07-24</td>
<td>ISSN 110—1836 No 190</td>
<td>3 (52)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Issued by</td>
<td>Mats Sandin</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4.5.2.3 Client........................................................................................................ 25
4.5.2.4 Object Request Broker (ORB) ........................................................................ 25
4.5.2.5 OMG IDL........................................................................................................ 26
4.5.2.6 ORB Interface ................................................................................................ 26
4.5.2.7 CORBA IDL stubs and skeletons......................................................................... 26
4.5.2.8 Dynamic Invocation Interface (DII)................................................................ 26
4.5.2.9 Dynamic Skeleton Interface (DSI)................................................................... 26
4.5.2.10 OLE Object Adapter....................................................................................... 26
4.5.3 Inside a CORBA application................................................................................ 27
4.5.4 How do remote invocations work? ...................................................................... 28

5 ANALYSIS .................................................................................................................. 30

5.1 MICROSOFT ........................................................................................................... 30
5.2 OBJECT MANAGEMENT GROUP ............................................................................. 30
5.3 SUN MICROSYSTEMS ............................................................................................. 31

6 THE APPLICATIONS ..................................................................................................... 32

6.1 EJB ......................................................................................................................... 33
   6.1.1 Application Servers and EJB Containers ......................................................... 35
   6.1.2 Packaging and Deploying the Bean .................................................................. 35
   6.1.3 Sun Microsystems’ Java 2 Platform Enterprise Edition .................................... 36
   6.1.4 Wide Specification.......................................................................................... 36
   6.1.5 Usage.............................................................................................................. 37
6.2 COM+.................................................................................................................... 37
   6.2.1 Platform and language..................................................................................... 38
   6.2.2 Object Pooling................................................................................................. 39
   6.2.3 Usage.............................................................................................................. 39

7 A DETAILED COMPARISON OF EJB AND COM+..................................................... 40

7.1 INDEPENDENT VS. PROPRIETARY ARCHITECTURE ............................................. 40
   7.1.1 A vendor independent architectural standard ............................................... 40
   7.1.2 A single vendor’s proprietary architecture .................................................... 40
7.2 AVAILABILITY ........................................................................................................ 40
   7.2.1 EJB .............................................................................................................. 40
   7.2.2 COM+.......................................................................................................... 40
7.3 SECURITY ............................................................................................................... 40
   7.3.1 EJB – COM+................................................................................................ 41
7.4 OBJECT POOLING ............................................................................................... 41
   7.4.1 EJB .............................................................................................................. 41
   7.4.2 COM+.......................................................................................................... 41
7.5 TRANSACTIONS ..................................................................................................... 42
   7.5.1 EJB .............................................................................................................. 42
   7.5.2 COM+.......................................................................................................... 42
7.6 LOAD BALANCING ............................................................................................... 42
   7.6.1 EJB .............................................................................................................. 43
   7.6.2 COM+.......................................................................................................... 43
7.7 SYNCHRONOUS VS. ASYNCHRONOUS ................................................................. 43
   7.7.1 EJB .............................................................................................................. 43
   7.7.2 COM+.......................................................................................................... 44
7.8 RESULTSET VS. RECORDSET ........................................................................................................ 44
7.8.1 EJB ....................................................................................................................................... 44
7.8.2 COM+ ................................................................................................................................... 44
7.9 REUSABILITY ............................................................................................................................ 45
8 CONCLUSION ................................................................................................................................ 46
9 SUGGESTIONS FOR FUTURE WORK ....................................................................................... 48
10 APPENDIX .................................................................................................................................. 49
11 REFERENCE LIST ......................................................................................................................... 51
2 Preface

First of all I would like to thank all the personnel, especially Emil Gustafsson, at NTier Solutions AB in Uppsala for making this master thesis possible and for the support I received during my work. I would also like to thank Roland Bol, at the Computing Science Department, for the encouraged and the feedback given to me during my master thesis.

During the first initial study of the technologies the focus was on getting a good idea of what they are, background and motives, facts, which where used later to determine which two technologies that should fall under a closer look and to be used in reference implementations.

Questions that will be answered are, e.g.:

- Scalability. What happens if my site becomes a top ten visited?
- Availability. Can I count on them to be there when I want them to?
- Security. Will my application be a target for intruders?
- Transaction policies. Is there a possibility that the bank will miss my last withdrawal?
- Load balancing. Which one of the many servers will have to do all the work?
- Reusability. Do I have to rewrite everything once again?

It was rather hard to get accurate information about the different technologies depending on what source used. Microsoft’s own papers and reports seemed to be too self uncritical to own products and too critical to other vendors products. Papers from independent sources seemed to be, perhaps, too critical too Microsoft products overall.

There are a number of abbreviations and definitions used in this report, which will be explained during its appearance. In general they appear in a certain context, inside a specific technology, thus getting a better understanding of those if explained under this context. However, there are three central concepts in the report that will be explained here: Objects, reuse and encapsulation.

- The word object tends to mean something different to everyone. In COM an object is some piece of compiled code that provides some service to the rest of the system. The difference to a source code object-oriented programming (OOP) object, such as those defined in C++ or Java, should be obvious. Such an object is defined as an instance of a class. For more information about technology specific objects, see under appropriate technology.
• *Reuse* can be divided into *Code reuse* and *Software reuse*. There are articles where the two completely separate techniques are used as synonyms. Code reuse is language specific, while Software reuse is language independent. The latter happens at the binary level. Code reuse is simply the process of having developers use the cut-and-paste strategy or as in some object-oriented languages such as C++ and Java, where the code is reused in inheritance or sub-classing for example. Software reuse is the process of writing routines once and wrapping them into some form of software, which later can be plugged into new applications.

• *Encapsulation* and *information hiding* are two other aspects that follow from the essence of component architecture. Even though many articles use the two techniques as synonyms, I will not. Encapsulation is the ability to collect proprietary information, such as attributes, methods and representation, at one place. Information hiding is the ability to hide, for example, the representation from the presentation. That is, how methods or data structures are implemented is irrelevant to the client.

Next a very short outline of chapter four to seven will be given:

• In chapter four, a pre study of the technologies will be done starting with COM, followed by DCOM, COM+, EJB and CORBA. It is here where the overall picture of the technologies will be explained, and the parts and design which they consist of.

• In chapter five an analysis of the technologies will be done, which will serve as the ground on what two technologies to choose.

• In chapter six a description of the application and the two implementations are given. Design decisions of the application and how they are implemented. Which tools are used and why.

• In chapter seven finally, a more detailed comparison of the two chosen technologies will be done. First an explanation is given of a certain issue used in both technologies, and thereafter a comparison of how the two technologies solved that issue.
3 Background

Before the client/server technology, the applications where developed as monolithic entities. I.e. no separation was made between accessing the data, presenting it to the user and the core of the application, the business logic. To this era the mainframe architecture and file sharing architecture belongs. With mainframe software architectures all intelligence is within the central host computer. Users interact with the host through a terminal that captures keystrokes and sends that information to the host. Mainframe software architectures are not tied to a hardware platform. A limitation of mainframe software architectures is that they do not easily support graphical user interfaces or access to multiple databases from geographically dispersed sites.

The original PC networks were based on file sharing architectures, where the server downloads files from the shared location to the desktop environment. The requested user job is then run (including logic and data) in the desktop environment. File sharing architectures work if shared usage is low. In the 1990s, PC LAN computing changed because the capacity of the file sharing was strained as the number of online user grew and graphical user interfaces became popular.

As a result of the limitations of file sharing architectures, the client/server architecture emerged. This approach introduced a database server to replace the file server. Using a relational database management system (DBMS), user queries could be answered directly. The client/server architecture reduced network traffic by providing a query response rather than total file transfer. It improves multi-user updating through a GUI front end to a shared database.

The two-tier client/server architectures have been built and fielded since the middle to late 1980s. With the two-tier client/server architectures, the user system interface is usually located in the user’s desktop environment and the database management services are usually in a server that is a more powerful machine that services many clients. The two-tier client/server architecture is a good solution for distributed computing when the size of the work groups are up to about 100 (As stated in [9]) people interacting on a LAN simultaneously. It does have a number of limitations. When the number of users increases, performance begins to deteriorate. This limitation is a result of the server maintaining a connection via "keep-alive" messages with each client, even when no work is being done. A second limitation of the two-tier architecture is that implementation of processing management services using vendor proprietary database procedures restricts flexibility and choice of DBMS for applications. Finally, current implementations of the two-tier architecture provide limited flexibility in moving (repartitioning) program functionality from one server to another without manually regenerating procedural code.
The three-tier architecture emerged to overcome the limitations of the two-tier architecture. In the three-tier architecture, a middle tier was added between the user system interface client environment and the database management server environment. There are a variety of ways of implementing this middle tier, such as transaction processing monitors, message servers, or application servers. The middle tier can perform queuing, application execution, and database staging. For example, if the middle tier provides queuing, the client can deliver its request to the middle layer and disengage because the middle tier will access the data and return the answer to the client. In addition the middle tier add scheduling and prioritization for work in progress. The three-tier client/server architecture has been shown to improve performance for groups with a large number of users (in the thousands) and improves flexibility when compared to the two-tier approach.

It is from here this master thesis will start. Introducing some of the technologies that have evolved, all focusing on software reuse, component architecture and so forth.
4 Pre Study of Technologies

4.1 COM

4.1.1 Overview

“The Component Object Model (COM) is a software architecture that allows applications and systems to be built from binary software components supplied by different software vendors. COM is the underlying architecture that forms the foundation for higher-level software services, like those provided by Object Linking and Embedding (OLE). OLE services span various aspects of commonly needed system functionality, including compound documents, custom controls, interapplication scripting, data transfer, and other software interactions.” [1]

COM is an object-based programming model designed to allow development of software components at different times by different vendors using a variety of languages, tools, and platforms.

4.1.2 Interfaces

In COM, applications interact with each other and with the system through collections of functions called interfaces. A COM interface defines the behavior or capabilities of a software component as a set of methods and properties. An interface is a strongly typed contract between software components that guarantees consistent semantics from objects that support it. An interface is the definition of an expected behavior and expected responsibilities. Interface names begin with “I” by convention, and each COM object must support at least one interface (called IUnknown see below), although it may support many interfaces simultaneously. An interface is not a class. While a class can be instantiated to form a component object, an interface cannot be instantiated by itself because it carries no implementation. Furthermore, different component object classes may implement an interface differently, so long as the behavior conforms to the interface definition (e.g. a stack implementation where one representation is an array whereas another is a linked list).

Component designers describe interfaces using Microsoft’s Interface Definition Language (Microsoft IDL), an object-oriented extension of the Distributed Computing Environment (DCE) Remote Procedure Call (RPC) Interface Definitions Language (IDL). Microsoft provides an IDL compiler that generates proxy and stub code in C or C++ from an interface definition. The generated proxy code provides a client-side application-programming interface (API) for objects that support the interface in question. Stub objects decode incoming client requests and deliver them to the appropriate object in the server. Internally, the proxy and stub code interact with the appropriate runtime libraries to exchange requests and responses. The COM runtime software is smart enough to bypass this extra work in the case of objects deployed in the same process as the client. It is also important to understand that a client does not have direct access to a COM object, but through a pointer to one of its interfaces. The pointer hides all aspects of internal implementation.
Interfaces are immutable. COMs interfaces are never versioned, which means that version conflicts between new and old components are avoided. A new version of an interface, created by adding more functions or changing semantics, is an entirely new interface and is assigned a new unique identifier.

4.1.2.1 IUnknown

Even though developers are free to define their own interfaces, each COM component must support one mandatory interface called IUnknown. IUnknown defines some essential functionality that provides the basic building blocks for managing object life cycles and allowing evolution of interfaces supported by an object. IUnknown has three methods: QueryInterface, AddRef and Release.

The QueryInterface method of IUnknown is the mechanism that allows clients to dynamically discover at run time whether or not an interface, specified as an interface identifier or IID, is supported by a component object. It is also the mechanism that a client uses to get an interface pointer from a component object.

The AddRef method of IUnknown is called when another component object is using the interface. The Release method is called when the other component no longer requires use of that interface. While the components object’s reference count is nonzero, it must remain in memory. When the reference counter becomes zero, the component object can safely unload itself from memory. Thus, the AddRef and Release methods are just reference counting methods.

4.1.3 Binary Interoperability

COM is an architecture that facilitates independent development of software components through a binary interoperability standard. This binary standard makes it easy to use off-the-shelf components without having access to source code. Once developed, COM components are easily deployed and integrated into a customer’s environment. COM is programming language independent and is provided on multiple platforms (e.g. Microsoft Windows, Apple Macintosh, UNIX). To that end, COM defines a binary call standard that dictates the layout of the call stack for all method invocations. The language independence is realized through a standard where COM defines a way to lay out *virtual function tables (vtables)* in memory, and a standard way to call functions through the vtables. All COM objects contain a vtable for each of their interfaces. The object also has a pointer to the vtable for each interface. When an interface pointer is returned to the client, it is in fact a pointer to a pointer. It points to the object’s vtable pointer for the interface. This is called double indirection. Basically this means that function calls between components are made through a pointer to a table holding the addresses to the called components functions. Thus, any programming language that can call functions via pointers (C, C++, Small Talk, Ada and even Basic) can be used to develop a COM component. In addition, COM provides mechanisms for communications between components, even across process and network boundaries, as we shall see under DCOM. It also provides mechanisms for shared memory management between components, error and status reporting and dynamic loading of components.
4.1.4 Globally Unique Identifiers (GUIDs)

COM uses *globally unique identifiers* (GUIDs). A GUID is a 128-bit integer used to identify every interface and every component object class. Component designers, using IDL, assigns a *universally unique identifier* (UUID), a term used instead of GUID as defined by the Open Software Foundation’s Distributed Computing Environment, to each interface to eliminate any ambiguity that might arise from name collisions. The identifier, called an interface identifier or IID, is also the cornerstone of COM’s interface versioning model. CLSID are GUIDs that refer to component object classes, and IID are GUIDs that refer to interfaces.

4.1.5 Packaging Transparency

There are two types of COM servers, depending on the memory address space where they are running. COM servers running at the address space of a client process are called *in-process servers*, whereas COM servers running at their own process address are called *out-of-process servers*. Physically, in-process servers are compiled as dynamic link libraries (DLL-files) whereas out-of-process servers are compiled as executables (EXE-files).

COM is designed to support *packaging transparency*, which means that developers can deploy components using several techniques. A COM server can be packaged as a DLL that is loaded into the client process when a class within the server is first accessed by a client. A COM server may also be packaged as a separate executable, that is an out-of-process server. This type of server can run on the same machine as a client or on a remote machine that is accessible using DCOM.

When an application wants to create an object, it uses COM to create that object based on a desired CLSID. This means that COM clients need not to be concerned with how a server is packaged or where its DLL or executable is located. The COM Library locates the implementation of a requested class and establishes a connection between the client and the server. This is done by requiring providers of COM classes to register information about the type and location of its DLL or executable in a local registry. COM’s Service Control Manager (SCM) has the job of looking up the CLSID in the registry and taking the appropriate action to activate the server. The COM Library uses SCM when asked to create an object.

4.1.6 Local/Remote Transparency

COM is designed to support *transparency*, which means that a client that invokes a request on an object will not be able to tell whether or not it is a local or remote call. The client code for interacting with an in-process server is identical to that for calling an out-of-process server, being on the same or different machine. Clients always call in-process code, meaning that component objects are always called by in-process code. COM provides the underlying transparent remote procedure call.
Client applications interact with COM objects through interface pointers. *Information hiding* provided by COM ensures that a client is not dependent on any implementation details of a COM object. In the case of in-process COM servers, calls made by a client using an interface pointer go directly to an object that is created in the client's process. Calls to objects in an out-of-process server go first to an in-process proxy object responsible for invoking the request using a remote procedure call. In the out-of-process server, a stub object receives each incoming call and dispatches it to the appropriate COM object. Figure 1 shows a COM client interacting with COM objects packaged in several different ways. Calls to local server processes are made through lightweight RPC, while calls to remote server processes are made through true RPC.

Figure 1:
4.2 DCOM

4.2.1 Overview

“The Distributed Component Object Model (DCOM) is a protocol that enables software components to communicate directly over a network in a reliable, secure, and efficient manner. Previously called “Network OLE”, DCOM is designed for use across multiple network transports, including Internet protocols such as HTTP.”[3]

Distributed COM extends the programming model introduced by COM to support communication among objects on different computers – on a LAN, a WAN, or even the Internet. These extensions include improved location and packaging transparency, additional threading models, new security options, and additional administration capabilities.

DCOM achieves the interprocess or intermachine communication through a proxy-stub mechanism with the help of a COM library. When a client application makes a call to a COM component that resides either in a different process on the same machine or on a different machine, the COM library (OLE32.dll) reads the configuration information from the system registry and creates a proxy object in the address space of the client application. It also creates a stub object in the address space of the server. The COM library communicates between these different processes or machines using either local procedure call (LPC) protocols or remote procedure call (RPC) protocols, depending on whether the client and the server reside on the same machine. The proxy object takes the call from the client, marshals the data and sends it across the process or machine boundary to the stub. The stub object unmarshals the data, initializes the server on the behavior of the client, and makes the actual call for the server to process. The COM library handles all these tasks transparently from the client and the server.
4.2.2 Location and Packaging Transparency

As we have seen above, COM applications are written independently of the packaging of the server. As being loaded into the client process as a dynamic link library (DLL) or launched as an executable in a separate process on the same machine. DCOM extends this transparency to include location transparency, allowing objects to exist anywhere on the network. The locations of the objects are completely hidden, whether it is in-process or out-of-process on a different machine. As with COM, the way that a client connects to a component and calls the component’s methods is identical.

DCOM "Object RPC" (ORPC) is based on DCE RPC. It extends RPC to include an object reference data type and adds a parameter to each call for the target object. When a client requests a COM class factory for a remote object, the local SCM contacts the SCM on the remote machine. The remote SCM locates and launches the server and returns an RPC connection to the requested class factory provided by the server. A class factory proxy is created for the client application, and the object creation continues as in the non-distributed case.
4.2.3 Free-Threading Model

Several factors impact the scalability of distributed object systems. To achieve the highest throughput on a single machine, developers typically need to build multithreaded servers. This type of object concurrency enables the server to leverage the power of a multiprocessor server machine. With free threading, a separate thread may handle each incoming object invocation, and multiple incoming calls can be dispatched on a single object at the same time in different threads. The COM library in conjunction with the RPC runtime library manages dispatching. The burden is on the developer to use concurrency primitives such as mutexes to ensure that all of the objects are thread-safe. For applications that need the highest performance, however, the added complexity pays off.

4.2.4 Security

The support of security is an important feature in all distributed systems. DCOM provides multiple levels of security through the use of Access Control Lists (ACL) on COM components. The ability to set the level of security provides another aspect on transparency for developers. The same binary component can be used in different environments with different security requirements.

4.2.5 Reference Counting and Pinging

Detecting client failures, catastrophic hardware failure or network crashes, in a distributed environment is one of many important issues an application developer is faced with. A common approach to this problem is to send keep-alive messages at periodic intervals that indicate that a process is still alive. This is particularly important with DCOM, given the use of reference counting to determine when an object should be removed. In order not to flood the network with keep-alive messages, DCOM uses keep-alive messages on a per-machine basis. That is, independent of the number of client processes on a machine, a single ping message is used between machines.

Additional optimizations available in DCOM include piggybacking ping messages onto ordinary requests and tracking changes to the set of remote references instead of always transmitting the entire set. Only if the entire client machine is idle with respect to a given server machine does it send periodic ping messages (at a 2-minute interval). To accomplish this tracking, DCOM implements a component called the Object Exporter, which tracks objects and object references that have been exchanged with other machines. The Object Exporter is responsible for tracking the aliveness of client processes on a given machine.
4.2.6 Load Balancing

Load balancing is an important feature when the number of clients using the application grows. DCOM does not have built-in, i.e. transparently, support for load balancing, even though it does make it easy to implement different types of load balancing.

4.2.6.1 Static Load Balancing

The least flexible method is called static load balancing. Static load balancing is when users are permanently assigned to a certain server running a certain application. Static load balancing requires an administrator and is suitable only when the load is predictable. In DCOM, changing a registry entry easily configures these settings.

4.2.6.2 Dynamic Load Balancing

Instead of permanently assign a user to a resource; a better choice would be to assign it to the most appropriate server available at that moment. DCOM managed this approach by using dedicated referral components. This component resides on a well-known server machine to which client components first connect, requesting a reference to the service they require. Using information about server loads, network traffic and statistics about past demands, the referral component can make a more suitable choice. After establishing the connection, the referral component withdraws and hence supports to a form of transparency.
4.3 COM+

4.3.1 Overview

“At the 1997 Professional Developers Conference in San Diego, CA, Microsoft announced plans for COM+, an extension to the Component Object Model (COM). COM+ builds on COM's integrated services and features, making it easier for developers to create and use software components in any language, using any tool. Delivered on the Microsoft® Windows® platform, COM+ is designed to preserve and extend developers' current investments in COM. Applications currently using COM technology will work in the COM+ environment.”[4]

COM+ has evolved from several Microsoft products: the Component Object Model (COM), Distributed COM (DCOM), Microsoft Transaction Server (MTS), as well as parts of Microsoft Message Queue (MSMQ). COM+ includes several middleware services beneath its hood, including transaction management, resource management, and security management.

COM+ is many things, but basically COM+ is the merging of the COM and MTS programming models with the addition of several new features. COM+ provides a much better component administration environment, support for load balancing and object pooling and an easier-to-use event model. MTS was designed to provide server-side component services and to fix some of DCOMs deficiencies, e.g., how it handles security issues, and the complete lack of a component management and configuration environment.

COM+ is based on binary components and interface-based programming. Method calls are remote across process and computer boundaries through the use of a transparent RPC layer. And just like COM components, COM+ components can be upgraded and extended in production without adversely affecting the client applications that use them.

COM+ supports distributed transactions and role-based security. It provides a built-in thread-pooling scheme that is as transparent as the one for MTS. The COM+ programming model also uses interception to expose platform services to developers through declarative attributes. However, COM+ takes attribute-based programming much further than MTS. In addition to transactional services and integrated security, COM+ exposes services such as custom object construction, synchronization, and object pooling. Other new COM+ features, such as Queued Components and COM+ Events, are also exposed through configurable attributes.

4.3.2 COM+ Catalog

As mentioned earlier, configuration information for COM (and MTS) is stored in the registry. COM+ introduces a database where this information will be stored, called the COM+ Catalog. This new database unifies the COM and MTS registration models and provides an administrative environment for components.
4.3.3 Load Balancing

With COM+ comes a way of transparently solving the problem with load balancing. COM+ provides a load balancing service that is transparent to client applications. To enable load balancing, one first has to define an application cluster, which is a series of machines (up to eight) having the server side components installed. Then a load balancing router machine has to be configured to route the creation request to one of the machines in the application cluster, as depicted in Figure 3.

Figure 3.

COM+ load balancing is implemented as an OS service that executes on the router node. As remote creation requests are received, it works with the local Service Control Manager (SCM) to route the request to the machine with the lightest load. Each component that is capable of load balancing has associated with it a load-balancing engine, which works with the router to determine which machine to route the request to. The default COM+ load-balancing engine uses a response time algorithm, where the response time is calculated by timing each method call through each interface on every instance on a given machine.
4.3.4 COM+ In-Memory Database – IMDB

COM+ uses a mechanism called in-memory database (IMDB) to enhance the scalability of applications. The IMDB is much like an ordinary cache mechanism where data is retrieved and stored locally. The IMDB is a robust, transient and transacted database system that operates only on physical memory. The IMDB implements a DB-oriented caching system that is optimized for lookups and retrievals. The IMDB can load tables from an existing back-end database or can house transient data as part of the application in use. This means that all requests for data from those tables will be handled on the application tier where Internet Information Server (IIS) and its components reside, eliminating a network trip for the data as depicted in Figure 4.

Figure 4.

Components access the IMDB using either its OLE DB provider interfaces or through ADO, which acts like a wrapper for OLE DB. The IMDB service is implemented on a per-machine basis and as far as I know, it does not support distribution. I.e. it is not possible to load the same table on multiple IMDB machines. In order for components to share information using the IMDB, they must execute on the same machine.

4.3.5 Queued Components

The COM architecture uses, as we seen, interfaces through which all communication passes. After a client connects to a component, all method calls are synchronous through the returned interface, which means that a client is busy until a call returns. RPC based calls are necessary for building distributed applications, meaning that the lifetime between the client and component is tightly coupled. The RPC connection is maintained as long as the interface pointer is valid.
In message-based, i.e. asynchronous communication, the client is not locked while waiting for a call to return. This gives the application developer additional flexibility in the areas of scalability and availability. If the component is not available, the client application can still execute due to the asynchronous communication. This separation of time means that the life cycles of the client and component is decoupled.

In COM+ this decoupling is achieved by using an underlying messaging system instead of direct RPC connections. In COM+ Queued Components, this underlying system is Microsoft’s Message Queue Server (MSMQ). If the component is not available, the system takes care of the retransmitting and waiting, leaving the client to still execute. This scenario is depicted in figure 5, with the MSMQ sitting between the client and server.
4.4  **EJB**

4.4.1 Overview

*Enterprise JavaBeans* is an open, standards based specification defining a server side Java framework for building and deploying Java components. An Enterprise JavaBean is specialized Java Class, developer written or generated by a tool, typically containing the business logic of a mult/tier distributed application. The basic characteristics of an Enterprise JavaBean are: “may be distributed over a network and accessed remotely, they are transactional and they are secure” [10].

Enterprise JavaBeans mandate a container model where common services are declared, not programmed. At development and/or deployment time, attributes defining the bean’s transactions and security characteristics are specified. At deployment time, the container, through introspection, determines the Enterprise JavaBean attributes for the runtime services it requires and wraps the bean with the required functionality. At runtime, the container intercepts all calls to the object and provides the transactional, threading and security behavior required before the method invocation, invokes the method on the object, and cleans up after the call – primarily a memory management function.

4.4.2 Two Types of Beans

The Enterprise JavaBeans specification specifies two types of enterprise beans, *session beans* and *entity beans*. An entity bean can be seen as a noun, representing a table in a database for example. While a session bean can be seen as a verb, representing the actions or methods used.

4.4.2.1 Entity Bean

An entity bean represents data persistently stored in a database and the methods for accessing and modifying that data. Entity beans are associated with database transactions, and may provide data access to multiple users. Since the data that an entity bean represents is persistent, entity beans survive server crashes (this is because when the server comes back online, it can reconstruct the bean from the underlying data.) In relational terms, an entity bean might represent an underlying database row, or a single SELECT result row. In an object-oriented database (OODB), an entity bean may represent a single object, with its associated attributes and relationships.

For entity beans, the choice of how a bean interacts with a database is left up to the developer. The EJB container can manage the interaction with the data store automatically (container-managed) or require the developer to write the interaction with the persistent store, usually a relational database (bean-managed).
Every EJB has a unique identifier. For entity beans, this unique identifier forms the identity of the information. For example, an EmployeeIDNumber might uniquely identify an Employee object. This is analogous to the concept of a primary key in a relational database system. A session bean’s unique identifier is whatever distinguishes it from other beans of its type: it may be the host name and port number of a remote connection, or even just a randomly generated key that the client uses to uniquely identify a given bean.

4.4.2.2 Session Bean

A session bean is an EJB instance typically associated to a particular client, i.e. it is usually not shared among multiple clients and as such contains the state of that particular client’s interaction. Session beans typically are not persistent (although they can be), and may or may not participate in transactions. In particular, session objects generally don’t survive server crashes. One example of a session object might be an EJB living inside a Web server that serves HTML pages to a user on a browser, and tracks that user’s path through the site. When the user leaves the site or after a specified idle time, the session object will be destroyed. Note that while this session object might be storing information to the database, its purpose is not to represent or update existing database contents; rather, it corresponds to a single client performing some actions on the server EJ Bs.

4.4.3 The Basic Part of an EJB System

Figure 6.
4.4.3.1 EJB Component

An Enterprise JavaBean is a component, just like a traditional JavaBean. Enterprise JavaBeans execute within an EJB container, which in turn executes within an EJB server. Any server that can host an EJB container and provide it with the necessary services can be an EJB server. (This means that many existing servers may be extended to be EJB servers, and in fact many vendors have achieved this, or have announced the intention to do so.)

An EJB component is the type of EJB class most likely to be considered an "Enterprise JavaBean." It's a Java class, written by an EJB developer, that implements business logic. All the other classes in the EJB system either support client access to or provide services (like persistence, and so on) to EJB component classes.

4.4.3.2 EJB Container

The EJB container is where the EJB component resides. The EJB container provides services such as transaction and resource management, versioning, scalability, mobility, persistence, and security to the EJB components it contains. Since the EJB container handles all of these functions, the EJB component developer can concentrate on business rules, and leave database manipulation and other such fine details to the container. For example, if a single EJB component decides that the current transaction should be aborted, it simply tells its container (in a manner defined by the EJB Spec) and the container is responsible for performing all rollbacks, or doing whatever is necessary to cancel a transaction in progress. Multiple EJB component instances typically exist inside a single EJB container.

4.4.3.3 EJB Object

Client programs execute methods on remote EJBs by way of an EJB object. The EJB object implements the remote interface of the EJB component on the server. The remote interface represents the business methods of the EJB component. The remote interface does the actual, useful work of an EJB object, such as creating an order form or deferring a patient to a specialist. EJB objects and EJB components are separate classes, though from the "outside" (that is, by looking at their interfaces), they look identical. This is because they both implement the same interface (the EJB component’s remote interface), but they do very different things. An EJB component runs on the server in an EJB container and implements the business logic. The EJB object runs on the client and remotely executes the EJB component’s methods.
4.5 **CORBA**

4.5.1 **Overview**

**CORBA** is the acronym for *Common Object Request Broker Architecture*, OMG's open, vendor-independent architecture and infrastructure that computer applications use to work together over networks. Using the standard protocols **IIOP** and **GIOP** together with the **OMG Interface Definition Language (OMG IDL)**, a CORBA-based program from any vendor, on almost any computer, operating system, programming language, and network, can interoperate with a CORBA-based program from the same or another vendor, on almost any other computer, operating system, programming language, and network. Using a standard architecture ensures portability and interoperability, two important properties in a distributed environment.

Using a CORBA-compliant ORB, a client can transparently invoke a method on a server object, which can be on the same machine or across a network. The ORB intercepts the call, and is responsible for finding an object that can implement the request, passing it the parameters, invoking its method, and returning the results of the invocation. The client does not have to be aware of where the object is located, its programming language, its operating system or any other aspects that are not part of an object’s interface.

The **OMA - Object Management Architecture** - categorizes objects into four categories: the **CORBAservices**, **CORBAfacilities**, **CORBAdomain objects**, and **Application Objects**. See figure 7.

Figure 7.
Benefits from using this model, according to OMG, reflects on applications as:

1. “Coding is quicker, so applications can be deployed sooner: Since your crew doesn't have to write as much code, development happens faster”. [12]

2. “Applications designed around discrete services have better architecture: Good architecture divides applications into modules or object groups based on functionality. By designing around the OMA, which is itself based on this principle, your application gets a head start on its own sound architecture”. [12]

3. “Many OMA implementations have enterprise characteristics built in: they're robust, and they scale: Providers know where you're going to deploy your applications with their OMA products, so they compete to meet your enterprise's needs. Scalable name servers, transaction services, and other services have proved their worth in enterprise deployments for years already”. [12]

4.5.2 Definitions

4.5.2.1 Object

This is a CORBA programming entity that consists of an identity, an interface, and an implementation, which is known as a Servant.

4.5.2.2 Servant

This is an implementation programming language entity that defines the operations that support a CORBA IDL interface. Servants can be written in a variety of languages, including C, C++, Java, Smalltalk, and Ada.

4.5.2.3 Client

This is the program entity that invokes an operation on an object implementation. Accessing the services of a remote object should be transparent to the caller. Ideally, it should be as simple as calling a method on an object, i.e., obj->op(args). The remaining components in Figure 8 help to support this level of transparency.

4.5.2.4 Object Request Broker (ORB)

The ORB provides a mechanism for transparently communicating client requests to target object implementations. The ORB simplifies distributed programming by decoupling the client from the details of the method invocations. This makes client requests appear to be local procedure calls. When a client invokes an operation, the ORB is responsible for finding the object implementation, transparently activating it if necessary, delivering the request to the object, and returning any response to the caller.
4.5.2.5 OMG IDL

OMG Interface Definitions Language is an object-oriented interface definition language used to specify interfaces containing methods and attributes. OMG’s IDL support interface inheritance (both single and multiple inheritance), and is designed to map onto multiple programming languages, e.g. C, C++, Java, COBOL, Modula 3, DCE, Smalltalk etc.

4.5.2.6 ORB Interface

An ORB is a logical entity that may be implemented in various ways (such as one or more processes or a set of libraries). To decouple applications from implementation details, the CORBA specification defines an abstract interface for an ORB. This interface provides various helper functions such as converting object references to strings and vice versa, and creating argument lists for requests made through the dynamic invocation interface described below.

4.5.2.7 CORBA IDL stubs and skeletons

CORBA IDL stubs and skeletons serve as the “glue” between the client and server applications, respectively, and the ORB. A CORBA IDL compiler automates the transformation between CORBA IDL definitions and the target programming language. The use of a compiler reduces the potential for inconsistencies between client stubs and server skeletons and increases opportunities for automated compiler optimizations.

4.5.2.8 Dynamic Invocation Interface (DII)

This interface allows a client to directly access the underlying request mechanisms provided by an ORB. Applications use the DII to dynamically issue requests to objects without requiring IDL interface-specific stubs to be linked in. Unlike IDL stubs (which only allow RPC-style requests), the DII also allows clients to make non-blocking deferred synchronous (separate send and receive operations) and one-way (send-only) calls.

4.5.2.9 Dynamic Skeleton Interface (DSI)

This is the server side's analogue to the client side's DII. The DSI allows an ORB to deliver requests to an object implementation that does not have compile-time knowledge of the type of the object it is implementing. The client making the request has no idea whether the implementation is using the type-specific IDL skeletons or is using the dynamic skeletons.

4.5.2.10 OLE Object Adapter

This assists the ORB with delivering requests to the object and with activating the object. More importantly, an object adapter associates object implementations with the ORB. Object adapters can be specialized to provide support for certain object implementation styles (such as OODB object adapters for persistence and library object adapters for non-remote objects).
4.5.3 Inside a CORBA application

CORBA applications are composed of objects, individual units of running software that combine functionality and data, and that frequently (but not always) represent something in the real world. Typically, there are many instances of an object of a single type - for example, an e-commerce website would have many shopping cart object instances, all identical in functionality but differing in that each is assigned to a different customer, and contains data representing the merchandise that its particular customer has selected.

For each object type, such as the shopping cart, one defines its interface in OMG IDL. This fixes the operations it will perform, and the parameters (input and output) for each. This interface definition is independent of the programming language in use, but maps to all of the popular programming languages via a set of OMG standards: OMG has standardized mappings for C, C++, Java, COBOL, Smalltalk, Ada, Lisp, Python, and IDL script.

This is the essence of CORBA - how it enables interoperability, with all of the transparencies claimed. The interface to each object is defined very strictly. But, in contrast, the implementation of an object - its running code, and its data - is hidden from the rest of the system behind a boundary that the client may not cross. Clients access objects only through their advertised interface, invoking only those operations that that object chooses to expose, with only those parameters (input and output) that are included in the invocation.

Figure 8 shows how everything fits together, at least within a single process: Compile the IDL into client stubs and object skeletons, and write the object (shown on the right) and a client for it (on the left). Stubs and skeletons serve as proxies for clients and servers, respectively. Because IDL defines interfaces so strictly, the stub on the client side has no trouble meshing perfectly with the skeleton on the server side, even if the two are compiled into different programming languages, or even running on different ORBs from different vendors.
Figure 8.

In CORBA, every object instance has its own unique object reference, an identifying electronic token. Clients use the object references to direct their invocations, identifying to the ORB the exact instance they want to invoke. The client acts as if it is invoking an operation on the object instance, but it is actually invoking on the IDL stub, which acts as a proxy. Passing through the stub on the client side, the invocation continues through the ORB (Object Request Broker), and the skeleton on the implementation side, to get to the object where it is executed.

4.5.4 How do remote invocations work?

Figure 9. Diagrams a remote invocation. In order to invoke the remote object instance, the client first obtains its object reference. To make the remote invocation, the client uses the same code that it used in the local invocation just described, but substitutes the object reference for the remote instance. When the ORB examines the object reference and discovers that the target object is remote, it marshals the arguments and routes the invocation out over the network to the remote object’s ORB.

Figure 9.
OMG has standardized this process at two key levels: First, the client knows the type of object it is invoking, and the client stub and object skeleton are generated from the same IDL. This means that the client knows exactly which operations it may invoke, what the input parameters are, and where they have to go in the invocation; when the invocation reaches the target, everything is there and in the right place. Second, the client’s ORB and object’s ORB must agree on a common protocol - that is, a representation to specify the target object, operation, and all parameters (input and output) of every type that they may use. OMG has defined this also – it is the standard protocol IIOP. (ORBs may use other protocols besides IIOP, and many do for various reasons. But virtually all speak the standard protocol IIOP for reasons of interoperability, and because it is required by OMG for compliance.)

Although the ORB can tell from the object reference that the target object is remote, the client cannot. (The user may know this also, because of other knowledge - for instance, that all accounting objects run on the mainframe at the main office.) There is nothing in the object reference token that the client holds and uses at invocation time that identifies the location of the target object. This ensures location transparency - the CORBA principle that simplifies the design of distributed object computing applications.
5 Analysis

Enterprise JavaBeans and COM+ are the two technologies chosen for further comparison. In this section a motivation will be given explaining on what grounds the decision is made. The intention from the beginning was to include the new Microsoft technology, DotNet, to this analysis, but due to time limitations it had to be excluded from the initial study.

5.1 Microsoft

COM is not a technology for distribution. COM was intended to be a component architecture running on a single computer. DCOM arrived as the bridge to accomplish remote invocation. Even though COM is the basis in both DCOM and COM+, it is excluded to the benefits of those two as being more evolved technologies.

DCOM relies on the Object Remote Procedure Call (ORPC) protocol. This ORPC layer is built on top of DCEs RPC and interacts with COMs run-time services. DCOM was brought to daylight as a protocol of letting COM break the barriers of intra communications across a network of interconnected computers. Even though it fulfilled its goal, a new project was launched at Microsoft to simplify the process and enhance the strength of the technology. Before COM+ entered the market, the MTS technology was incorporated into COM, thus given the functionality and strength requested. Still, using MTS a wrapper around COM did not decrease the effort a developer had to face during a distributed implementation. Instead COM+ entered the scene. A full fledge technology for distribution of components in computer networks. In addition, increased functionality and attribute based programming thus making it easier for the developer. COM+ is somewhat Microsoft top technology for distributed programming, even though DotNet is expected to take over that role.

5.2 Object Management Group

CORBA relies on the Internet Inter Protocol for remoting objects. Everything in the CORBA architecture depends on an Object Request Broker that acts as a central bus over which each CORBA object interacts transparently with other CORBA objects. This is essentially what the CORBA specification says today. A new specification is under construction from the OMG, specifying component architecture as an extension to the present specification. Even though I have not seen the specification, the idea sounds interesting and could probably stand up to COM+ and EJB better than today.
5.3 Sun Microsystems

The Remote Method Invocation of Java relies on the Java Remote Method Protocol. To invoke RMI, every object has to be marshaled as a stream through the Java Object Serialization thus restricting the objects, both servers and clients, to be written in Java. Even though Enterprise JavaBeans relies on RMI, and thus restricted to Java as implementation language, the EJB specification conforms to being an object architecture allowing objects to be distributed across a heterogeneous network.
6 The Application

First of all I want to point out that these two applications stand as basis for my master thesis, and should therefore be seen just as reference implementations. The design decision is essentially the same for both the EJB application and the COM+ application. The design decision is made out of mainly two factors:

- Using a multitier solution, with intercommunication between different components, lead to an application with more then one component.

- The company, NTier Solutions AB, which hosted me during my work had already due to an earlier master thesis, an existing database interface implemented in COM. Even though the design was made on this factor, the existing interface was discarded and completely rewritten.

Figure 10.

In the implementation phase, Traci played a great role when functionality of the two techniques was tested. Traci is a time report system used internally by NTier Solutions AB. Data is stored in a relational database (Microsoft SQL Server) together with some logic in form of stored procedures. Access to data is done through procedure calls from a data access layer. To obtain a higher abstraction in the data access layer, the argument passed to the database was converted to a string. Thus making no distinction whether the request is done to a stored procedure or a regular SQL statement. The middle layer functions only as a bridge between the representation layer and the data access layer, i.e. it passes the argument given from the representation layer to the data access layer and vice versa with the obtained data. Data is delivered to the presentation layer, which modifies the data and delivers it as a HTML-formatted string to the application. Notice that the HTML-formatted string could easily be adapted to fit a WAP application or an ordinary stand-alone application.
6.1 EJB

The application consists of mainly two parts, the two Enterprise JavaBeans representing the backend of the application (see figure 11). Because the role of these beans is to define the core business logic of the application, some other way was needed to deal with the presentation issues like generating web pages to communicate results. Though the plan was to make this a web-based application around Enterprise JavaBeans, the architecture was broken up into two main areas:

- The business logic (in this case database connection and processing), provided by Enterprise JavaBeans
- The presentation logic (the logical connection between the user and EJB services), provided by a Java Servlet. Here the Servlet is used to produce the actual HTML output.

As a third part one could make use of Java Server Pages (JSP) to the presentation layout (the actual HTML output). But though I had not the time to familiarize myself with the JSP technology, I skipped this part. Note that the JSP technology is just an extension of the Servlet technology. Every JSP is compiled into a Servlet during runtime.
As mentioned earlier, there are two primary types of Enterprise JavaBeans – session and entity beans. Session beans provide a service, usually work with objects passed in as parameters, and generally disappear between transactions. Session beans take their name from the fact that they are usually instantiated by the client and exist only for the duration of a single client-server session, where the session bean performs operations on behalf of the client.
There are two types of session beans – stateless, the most common type, and stateful. Stateless beans simply perform a service and do not have any persistent data, whereas stateful beans maintain some data between requests but are generally short lived. Stateful beans are responsible for handling their own data persistence issues.

With this in mind the choice of two stateless session beans was rather straightforward. One bean, which acts like a simplified interface abstraction and a second bean which in a real application would contain the business logic, but here serves as a forwarder.

6.1.1 Application Servers and EJB Containers

Enterprise JavaBeans work in concert with an EJB container, typically integrated into an application server. EJB containers must support Sun’s Enterprise JavaBean specification, which details the interface between beans and other application server elements. Enterprise JavaBeans can be used with any application server or other system providing an EJB container that implements these interfaces. EJB containers can also exist in other systems such as transaction monitors or database systems.

The EJB container also provides a communications channel to and from its beans, and it will handle all of its beans’ multithreading issues. In fact, the EJB specification explicitly forbids a bean from creating its own threads. This ensures thread-safe operation and frees the developer from often, complicated thread management concerns.

6.1.2 Packaging and Deploying the Bean

To deploy an EJB component means to load it into our application server’s container. Beans are package into Java archive (JAR) files along with a special version of the JAR file’s manifest file (essentially its table of contents) and a serialized instance of a deployment descriptor object, which tells the container how to interact with the bean. The deployment descriptor specifies how the bean is supposed to be used in this particular application. For example, it might dictate who has access to the bean, the specifics of starting and concluding transactions, and such. While the contents of a bean package are prescribed in the EJB specification, the actual tools that package and build deployable beans are left to the design of the application server developers. Most application server developers will provide tools you can use to build your deployment descriptor.
6.1.3 Sun Microsystems’ Java 2 Platform Enterprise Edition

There are a variety of application servers on the market today with cost range from completely free to thousands of dollars. One of the primary goals of the EJB standard architecture is to provide portability over not just operating systems but over application servers as well.

“The Enterprise JavaBeans server component model defines a standard architecture for Java applications servers. The primary goals of EJB are to enable portability across application servers, component reusability, and increased developer productivity.” [10]

The *Java virtual machine (JVM)* allows a Java application to run on any operating system, but server components require additional services that are supplied by an application server rather than by the JVM. I.e. any EJB compliant component can run in any EJB compliant application server.

One of the intentions was to test whether or not this goal is achieved. Even though it has not been done due to reasons described above and due to time overhead of retrieving and configuring another server, I cannot see any problem why this should not work. The reason is the *J2EE Compatibility Test Suite*, which is a deliverable included in the Java 2 Platform Enterprise Edition. A middleware vendor must successfully pass the test suite to receive the J2EE certification logo. This ensures that J2EE application code can be developed once, compiled, and deployed into any j2EE platform-based product.

Sun Microsystems’s Java 2 Platform Enterprise Edition (J2EE) reference server is used in the examination work of the master thesis. It is free of charge and available for downloading from Sun’s home page [10]. In many of those examples and tutorials studied before the actual implementation, the J2EE was heavily used and therefore easy to adopt. In addition it is rather easy to find external documentation on how to install, configure and use as well.

6.1.4 Wide Specification

The wide range of the Enterprise JavaBean specification leads to a harder task of getting a deeper understanding of the technology. This should not be any problem for a big corporation that can satisfy the potential of the EJB specification, and which has the ability to maintain a broad internal competence. For a smaller company though, the fact that it is a challenging task just to get a full understanding of the technology can lead to problems. It is always a cost to have personnel up to date with new technologies, even if it pays off in the end. If however, a company can do without the full potential, it is a good idea to adopt the technology and get an understanding on the way.
6.1.5 Usage

Thanks to the many tutorials residing on the Internet and the many books available, the EJB technology is rather easy to get started with. The problem is not to find the information; it is to extract the wanted one.

6.2 COM+

As in the case with the EJB application, this application also consists of mainly two parts, the two COM+ components representing the backend of the application (see figure 12). The presentation logic consists of an Active Server Page, ASP, which produce the actual HTML output showed in the client browser.

Figure 12.

To interact with any kind of data stores, including relational database systems, flat files and emails, Microsoft uses a strategy called Universal Data Access (UDA). As an important part of this strategy, Microsoft’s Data Access Components (MDAC) supports three major technologies: OLE DB, Microsoft ActiveX Data Objects (ADO) and the Open Database Connectivity (ODBC). The first technology, OLE DB, is a low-level, high-performance interface to a variety of data stores. The second technology, ADO, is a high-level, easy-to-use interface to OLE DB. The third technology, ODBC, is another low-level, high-performance interface that is designed specifically for relational data stores.
There are a variety of approaches when using ADO as the technology for interacting with a relational database. ADO allows you to use the Command Object without explicitly creating a Connection object, by setting the ActivConnection property to a connection string. There are a couple of downsides to using this approach. First, ADO automatically creates a new Connection object instead of using an existing connection, thus consuming additional server resources. Second, the Connection object has an associated errors collection, which stores an OLE DB provider error during the session. The error collection is available only through the explicitly created Connection object. Notice that the ADO Connection object is explicitly used in this work and set as the ActiveConnection property, thus eliminating the downsides described above.

### 6.2.1 Platform and language

A COM+ application consists of COM components that are compiled as ActiveX DLLs. A COM+ application is roughly equivalent to an MTS package in Windows NT 4.0. A COM+ application allows the COM components it contains to receive a richer set of services provided by Windows 2000 Component Services. Therefore, Microsoft’s Windows 2000 is used as the platform for this implementation, with its integrated Internet Information Server (IIS) and the Active Server Pages technology (ASP). IIS and ASP provides the ability to produce the dynamic and interactive Web page used in the presentation layer.

To build Windows DNA 2000 applications using COM+ services any programming language that is COM compatible, such as Microsoft Visual Studio tools, can be used. Here the choice became Visual Basic 6.0 due to its integrated support for Component Development.

When compiling an object written in Visual Basic all the GUIDs are generated automatically, including CLSIDs, IIDs and type library Ids. Some programming language, such as C++ and Java, separate the interface definition from the classes. They define the interface through the Interface Definition Language (IDL), save it as a separate file, and then compile the IDL file to generate source files in Visual Basic however, this procedure is all handled behind the scenes.

As mentioned earlier, the COM specification requires that all COM components must support the IUnknown interface with its three methods: QueryInterface(), AddRef() and Release(). This support is also handled behind the scenes by Visual Basic. Thus examining a DLL-file produced by Visual Basic through the Visual Studio Tool, OLE View reveals the implicit defined interfaces.
6.2.2 Object Pooling

COM+ provides the object pooling service to reduce the overhead incurred in object creation by maintaining a pool of objects instead of destroying and re-creating object each time after using them. For the object to be pooled, it must meet several requirements. For example, the object must have no thread affinity. COM+ supports multiple *threading models* for components. A threading model defines what threads a component’s object can run on and how COM+ will synchronize access to the objects. All threading models are based on the notion of *apartments*. An apartment is an execution context for objects. Every object resides in exactly one apartment for its entire lifetime. One or more apartments reside within each process. To make use of object pooling, the object must support *Multi Threading Apartment (MTA)* or free threading model. Visual Basic 6.0 supports only *Single Apartment Threading (STA)* so the use of VB 6.0 in the application lead to an absence of pooled objects.

6.2.3 Usage

COM+ is an easy technology to find information about. The MSDN provides tutorials, which can help developers to get started. There are several books available. Most of which are describing the COM+ technology in C++. I found it rather frustrating to getting started with the new IDE environment though. But that is just a gap to overcome once; thereafter it is essentially straightforward. It requires more time than a few weeks of tutorials and tests to get a full grip of the strength of COM+. The management tools provided are subjects to questions, probably easy to use when fully understand.
7 A Detailed Comparison of EJB and COM+

7.1 Independent vs. Proprietary Architecture

7.1.1 A vendor independent architectural standard

J2EE is an open architecture that represents the combined expertise of a collaborative industry effort, and has wide industry support from key middleware vendors. This implicates that customers have the ability to select products from an ecosystem of server side platforms, tools and components. It promotes competition among vendor products and tools, and thereby promotes best-of-breed products.

The wide range of application servers on the market today leads to greater flexibility where internal issues are of importance, such as e.g. politics, security.

7.1.2 A single vendor’s proprietary architecture

In comparison, Microsoft’s COM+ is based on proprietary technology, raising barriers to entry that prevent competition. It is a single vendor technology, which binds applications to Microsoft’s windows and prevents other vendors from providing windows implementations.

7.2 Availability

A company, which relies on its web-applications, can suffer in significant lost business during a planned downtime. Unplanned downtime during peak hours can be disastrous. Today’s most robust operating systems can reach levels of 99.999% availability, or as little as 5 minutes of downtime per year.

7.2.1 EJB

The J2EE platform is an open architecture with implementations available for several different operating systems. This gives the customer freedom to select an operating system, which satisfies the customer’s need of robustness and availability.

7.2.2 COM+

In comparison, Microsoft products are tightly coupled to Microsoft Windows, which historically has been unable to reach the highest point of availability. Even though Microsoft has been touting Windows 2000 as highly available, I have faced the fact of rebooting the system a few times.

7.3 Security

Security is often divided into two areas, Authentication and Authorization. Authentication of clients is the process of testing whether clients are who they claim to be. Authorization of clients is the process of testing whether clients have access rights to perform desired tasks.
7.3.1 EJB – COM+

The two technologies make use of quite the same techniques. They both authenticate clients over *Secure Socket Layer (SSL)*. JSP and Servlets in EJB and ASP and ISAPI code in Windows DNA. Applications code can access credentials in any naming and directory system via JNDI in EJB, or in Microsoft’s Active Directory for COM+. Authorization of components can be controlled either programmatically or declaratively in both technologies.

7.4 Object Pooling

Instead of physically destroying and/or recreate an object each time the client releases a reference to the object; the system can place the object in a pool. The next time the client or another client wants to use the object, the system retrieves one from the pool that is not used. This process, which is called *Object Pooling*, is completely transparent from the client. Even though this technique involves some overhead with getting an object from the pool, reusing live objects instead of reinstantiating them from scratch, reduces the overhead to a certain extent.

7.4.1 EJB

The EJB container hosting the bean, not by the application, controls the lifecycle of a bean. After instantiation, the entity bean moves to a pool of available instances. While in the pooled stage, the instances in the pool are identical. The EJB container assigns an identity to an instance when moving it to the ready stage. In the pooled state, an instance is not associated with any particular EJB object identity. With bean-managed persistence, when the EJB container moves an instance from the pooled state to the ready state, it does not automatically set the primary key. Therefore, the *ejbCreate* and *ejbActivate* methods must set the primary key. If the primary key is incorrect, the *ejbLoad* and *ejbStore* methods cannot synchronize the instance variables with the database.

7.4.2 COM+

Microsoft’s Visual Basic 6.0 does not support object pooling due to its current threading model used. Visual Basic 6.0 supports only the Single Threaded Apartment model (STA). The STA model requires that all the calls to the object must be made through a single thread in the main apartment. This is called *thread affinity*. In contrast, to make use of the object pooling, the object itself must not be tied to any specific threads. The object should be accessible by any threads. This requires that the object must be a free threaded model.
7.5 Transactions

A transaction is a set of operations that is treated as a unit of work, i.e. a transaction has to be completed in its entirety or not at all. Every transaction must guarantee four properties, called ACID:

- **Atomicity** – A transaction can either commit or abort its changes
- **Consistency** – After completion, a transact guarantees the state of changes to be reliable and consistent
- **Isolation** – Concurrent transactions are isolated from modification made by other incomplete transactions
- **Durability** – When transaction commits, its changes become permanent

7.5.1 EJB

A transaction in EJB can be controlled programmatically or declaratively. Programmatically controlled transactions are done through the Java Transaction API (JTA). Declarative transactions can be specified through the deployment tool at a component method level, which means that each method in an EJB component can perform transactions differently. Every J2EE platform must support automatically handled transactions, thus leaving the developer to focus on the business problem at hand. Even if distributed transactions are optional in the J2EE specification, many vendors’ implements this service as well.

7.5.2 COM+

As in EJB, transactions can be controlled programmatically or declaratively. Programmatically controlled transactions are done through the OLE Transactions. Declarative transactions can be specified through the Component Services tool. Declarative transactions are somewhat limited compared to EJB because different methods within a component interface cannot have different transactional characteristics. Distributed transactions are available through the Microsoft Distributed Transaction Coordinator (MS DTC).

7.6 Load Balancing

Load balancing is dividing the amount of work that a computer has to do between two or more computers so that more work gets done in the same amount of time and, in general, all users get served faster. To meet an increasing traffic, a corporation must be able to scale any given tier by adding additional computers to that tier.
7.6.1 EJB

A component contained in a J2EE container automatically benefit from underlying load balancing. No special code fragments have to be written, nor are any extra binaries needed. J2EE is an open architecture that has wide support from key middleware vendors. As a result, it promotes competition between vendor product and tools. Today, many vendors implement a wide variety of algorithms for load balancing. Among these we have Round Robin, random and dynamic load balancing. The result is a choice of a wide variety of solutions that offer strong scalability features.

7.6.2 COM+

Today, Windows DNA support a load balancing technology called Network Load Balancing. Network load balancing is used to balancing incoming requests to Web server machines. Microsoft announced that a technology called Component Load Balancing (CLB) was to be included in the first specification of COM+. CLB was intended to balancing the traffic from the clients to the business tier but due to lack of functionality, Microsoft decided to discard the CLB. Therefore, COM+ does not currently offer load balancing on component level.

7.7 Synchronous vs. Asynchronous

An application that uses synchronous communication, e.g. RPC, sends a request to a server and then waits for an answer to come back, thus the client is blocked and wasting valuable time of doing nothing. In the case of an absent response, the client could be blocked forever or until a mechanism frees the client. A better choice would be to send a request, continue with other tasks until a response comes back and then handling that response. In fact, the sender does not need to know anything about the receiver, nor does the receiver need to know anything about the sender, they only need to know what message format and what destination to use. This is called asynchronous communication, and enables distributed communication that is loosely coupled. A message sent is first stored into some kind of queue before the communication system passes the message to the callee. If the callee is not available at the moment, the message is kept in the queue until the callee is up again.

7.7.1 EJB

The Java Message Service (JMS) is a Java API that allows applications to create, send, receive and read messages. Any EJB component can send or synchronously receive a JMS message. Since I started the study of the Enterprise JavaBean component architecture, a new kind of enterprise bean has evolved. The message-driven bean which, enables the asynchronous consumption of messages. Optionally, a JMS provider may implement message-driven beans to process messages concurrently. Unfortunately, there is no time to examine this new enterprise bean, thus leaving this topic open with the sense that messaging in EJB is possible in one way or the other.
7.7.2 COM+

The COM+ Queued Components (QC) service extends the asynchronous programming model of MSMQ to the component level. QC is one of the most important COM+ component services. By using QC, a client is no longer blocked by the server process after making a call because the call is now processed asynchronously. As a result, scalability is improved. Moreover, QC allows clients to make the call even when the server components are not available. This capability makes enterprise applications more robust.

7.8 ResultSet vs. Recordset

Querying a database often results in a returned set of data containing, for example, tables, which are a subset of those residing in the database. In Java JDBC this data set is called a ResultSet, in Microsoft’s ADODB it is called a Recordset.

7.8.1 EJB

When working with Enterprise JavaBeans there is no way of passing the ResultSet, without first converting it to a new representation, between objects over the network or through layers of business objects. The reason is the tight coupling between the connections used, i.e. the lifecycle of the ResultSet. A ResultSet object is created through the getResultSet call, and is automatically closed when the Statement object that generated it is closed, re-executed, or used to retrieve the next result from a sequence of multiple results. There are mainly two things obtained through this behavior: abstraction and memory management.

Because the ResultSet cannot be returned without first being transformed into another object, the caller is unaware of what kind of storage the requested data is retrieved from and therefore enhanced abstraction is obtained. The other behavior, memory management, is obtained through a mechanism, which read as much data from the database as needed. As the cursor moves through the ResultSet, sufficient amount of data is fetched from the database in a transparent manner.

7.8.2 COM+

Working with Microsoft’s ActiveX Data Objects present one main advantage over the Enterprise JavaBean technology, disconnected Recordsets. A disconnected Recordset does not hold any connections to the database, which means a more effective use of resources. Keeping connections open unnecessarily reduces the number of connections allowed depending on current access licenses. The Recordsets can be passed between objects over the network or through layers of business objects.
7.9 Reusability

Even though EJB is a component software technology, I felt during my work that COM+ is more mature in the sense of being a component architecture. A COM component is developed, registered as a Microsoft DLL-file and deployed into a COM+ environment where other software components can make use of its functionality. In EJB, even though it also signs up to be component architecture, it is closer in hand to reuse it as packages as in ordinary Java development.
8 Conclusion

Due to the open specification and many vendor solutions the EJB feels more comfortable in the context of safety critical systems. Even though Microsoft Windows has evolved, it still does not feel stable. In contrast, letting the customer choose platform, EJB offers greater flexibility and enhanced stability.

Both EJB and COM+ provide good abstraction when designing middle tier components. The experience is that COM+ is more mature for component reuse. Once a component is written and registered, it is available for use. Since the component remains in the system, it can be referenced by other newly developed components as easily. EJB has a more complicated approach due to its required EJB container. Every EJB component has to reside inside an EJB container, thus making the deployment of the components a little bit more complicated. The idea of reusing a component is the same, but in the EJB case, every newly developed component has to deploy. Once deployed, it functions can be used as in the case of COM+. Still, the fact that is a more complicated task to deploy an EJB component than a COM+ remains. This is perhaps due to my use of the Sun Microsystems J2EE reference server. A commercial server like Beas WebLogic possibly eliminates these difficulties.

Even though COM+ is language independent, I am not sure that it is an advantage over EJB and Java. In a perfect world, every language is used in its specific area. Visual Basic for GUIs, C++ in performance critical systems and Prolog in systems with advanced logics. In most cases though, I feel that it is better to standardize on a single language. Using one specific implementation language during implementation phase of a software engineering project enhances the support and communication inside the project, thus increases the productivity. Increase of maintainability is another aspect. Having developers coding in different languages during a project does not really increase the productivity. Using Microsoft Visual Studio, with Visual Basic, does inflect on development, though. With its built-in support for COM development it suits many developers requirements well. With its ease to use and short time to market, it is widely used for component development inside the domain of Microsoft products.
My guess is that Microsoft’s Windows feels more comfortable for most people. It is the far most used operating system today for personal computers. Its environment is incorporated into all Microsoft products, an important issue in human computer interaction, thus gives developers higher confidence for their work. A corporation that uses Microsoft products should, in combination to the above statements and with respect to earlier conclusions, consider the COM+ solution. It is somewhat more straightforward, more an “all under the same roof” solution so to speak.

In the end when it comes to which architecture to choose, I will recommend concentrating on the larger business issues. Have in mind the existing developer skill sets, the existing system of the corporation in mind, the existing vendor relationships, and the customers. Regardless of which technology chosen, new developers will need to be trained.


9 Suggestions for Future Work

With the introduction of Microsoft .Net (DotNet), the new component architecture from the OMG and the new EJB specification, the evolution of component software architecture continues. A new language, C# (C Sharp), has been defined by Microsoft to meet the needs and requirements from .Net. The specification from the OMG is the youngest of the above technologies, thus the least mature. Even though .Net is a new technology, Microsoft has years of experience in component architecture.

With this information in mind, some interesting new openings arise:

- Why .Net?
- Is .Net an extension of the COM technology or a complete new architecture?
- The migration from the previous platform to the new platform?
- Similarities between OMG and Microsoft/EJB?
- Differences between OMG and Microsoft/EJB?
- How will the new features in the EJB specification affect the use of it?
10 Appendix

Some changes and some new features have showed up since the start of the Master Thesis. Topics which I briefly will try to explain in this section. Interested readers are encouraged to visit the corresponding Web sites for more information.

In the COM+ 1.0 beta version, terms like Component Load Balancing (CLB) (See section 6.3), In Memory Database (IMDB) (See section 6.4) and Transactional Shared Property Manager (TSPM). In the final release of COM+ and Windows 2000 they are all gone. CLB and IMDB are explained earlier in the report see sections above. TSPM is a programming interface that creates a cache for a client to use for transactional, synchronized access to application-defined shared properties across multiple processes. TSMP is built on top of IMDB. TSMP uses IMDB to persist data.

The EJB 2.0 specification introduces a new type of enterprise bean, called a message-driven bean, in addition to session beans and entity beans. Session and entity beans are very much the same as in the earlier EJB architecture. However, there have been changes to how these beans are accessed and used.

In brief, the EJB 2.0 specification introduces these new features:

- Message-driven beans
- An improved architecture for container-managed persistence
- Container-managed relationships for entity beans with container-managed persistence.
- Local interfaces
- Enterprise JavaBeans Query Language

Message-driven beans are stateless, server-side, transaction-aware components that process asynchronous messages delivered via the Java Message Service (JMS). Message-driven beans serve as router processes that operate on inbound enterprise messages from a JMS provider. A message-driven bean might be used to integrate an EJB-based system with a legacy system or to enable business-to-business interactions. The message-driven bean's sole responsibility is processing messages, because its container automatically manages the component's entire environment.
Session beans and entity beans in 2.0 can have two types of interfaces; a remote interface and a local interface. The local interfaces for session and entity beans provide support for lightweight access from enterprise beans that are local clients. That is, the bean uses a local interface if the bean wants to provide tight coupling with its clients with pass-by-reference semantics. The local interface is a standard Java interface that does not inherit from RMI. A bean uses the local interface to expose its methods to other beans that reside within the same container. It is thus possible to directly access a bean through its local interface without the overhead of a remote method call.

However, one should have in mind that the EJB 2.0 specification is not in its final form, and there could be further changes.
11 Reference list

  <URL: http://msdn.microsoft.com>


  <URL: http://www.microsoft.com/com/tech/dcom.asp>

  <URL: http://www.microsoft.com/com/tech/complus.asp>

  <URL: http://www.microsoft.com/dna>

  <URL: http://www.microsoft.com/data>

  <URL: http://www.microsoft.com/msmq/default.htm>

  <URL: http://www.asptoday.com>

  <URL: http://news.dci.com/geos/dbsejava.htm>

  <URL: http://www.sun.com>

  <URL: http://www.javaworld.com>

  <URL: http://www.omg.org>


