Dependability aspects of COM+ and EJB in multi-tiered distributed systems

(HS-IDA-EA-02-108)

Róbert Kárason (b99robka@student.his.se)
Department of Computer Science
University of Skövde, Box 408
S-54128 Skövde, SWEDEN

Final Year Project in Computer Science, spring 2002
Supervisor: Sanny Gustavsson

Dependability aspects of COM+ and EJB in multi-tiered distributed systems
Submitted by Róbert Kárason to Högskolan Skövde as a dissertation for the degree of B.Sc., in the Department of Computer Science.

6 June 2002
I certify that all material in this dissertation which is not my own work has been identified and that no material is included for which a degree has previously been conferred on me.
Signed: _____

Dependability aspects of COM+ and EJB in multi-tiered distributed systems

Róbert Kárason (b99robka@student.his.se)

Abstract

COM+ and Enterprise JavaBeans are two component-based technologies that can be used to build enterprise systems. These are two competing technologies in the software industry today and choosing which technology a company should use to build their enterprise system is not an easy task. There are many factors to consider and in this project we evaluate these two technologies with focus on scalability and the dependability aspects security, availability and reliability. Independently, these technologies are theoretically evaluated with the criteria in mind.

We use a 4-tier architecture for the evaluation and the center of attention is a persistence layer, which typically resides in an application server, and how it can be realized using the technologies. This evaluation results in a recommendation about which technology is a better approach to build a scalable and dependable distributed system. The results are that COM+ is considered a better approach to build this kind of multi-tier distributed systems.

Keywords: COM+, EJB, distributed systems, multi-tier, components, dependability, scalability.

Table of contents

Table of figures

1 Introduction

When writing software, it is possible to start from scratch and maybe be forced to code the same piece of functionality more than once. Another way is to use components, and services provided by the underlying operating system. This means that an application programmer can focus her efforts on the business logic and developing user interfaces. Examples of technologies to build component-based applications are COM+ (Microsoft Corporation, 2002b) and EJB (Sun Microsystems, 2001), which are the ones covered in this report. However, these techniques have diverse support for different criteria, and choosing the appropriate technique can be difficult (Sessions, 2000a), because different companies have different requirements

In this report we evaluate how COM+ and EJB supports the dependability aspects defined by Laprie (1994), such as security and availability, and apply this evaluation to a real world problem that faces Theriak[1], a software development company that deals with software systems for medication management within hospitals and other healthcare institutions and organizations. Their current product does not, in their opinion, have enough support for issues like dependability, therefore the company is moving towards building component based application, using either COM+ or Java based technologies.

Furthermore, in order to increase support for scalability a 4-tier distributed system is used in the evaluation. This architecture is suggested by Theriak, and one of the reasons that the company is going into this 4-tier distributed system is scalability. This is because it is hard to estimate the number of clients in healthcare organizations at the time of construction.

In the design phase, alongside architectural and system design, choosing the right technique is vital. Starting off on the wrong foot might turn out to be disastrous in the implementation phase. This means that companies need to evaluate COM based technologies and Java based technologies (EJB) based on certain prioritized criteria. The biggest difference between COM+ and EJB, according to Sessions (2000a) is that while EJB has greater support for platform independence, COM+ has greater support for performance (in a Windows environment) because it is built for that platform. Although these are important facts to consider, it is also necessary to look at what options these technologies have to offer for the application programmer to implement a middle tier in a distributed system.

Based on the architecture suggested by Theriak, this report aims to point out the advantages and disadvantages of COM+ and EJB, looking at the techniques with dependability issues in mind. This results in a recommendation for a company like Theriak about what technique they can choose to build their software.

[1] http://www.theriak.is

1 Introduction

This report begins with a background of related terminology, which describes terms like distributed systems, middleware, COM+, EJB and so on. These terms are necessary as a preparation for the problem domain. Chapter 3 contains the problem definition where the problem is introduced and discussed. Chapter 3 also introduces the system architecture suggested by Theriak and a discussion about the evaluation criteria, and the project delineations.

After defining the background terms and the problem, there is a discussion about appropriate methods to solve the problem. The chosen method is presented and arguments for using this method are given.

Realization of the method is the next part of the report. This section describes COM+ and EJB in more depth, and how these technologies handle the criteria.

Lastly, there are the results, where there is a discussion about the recommended technique and arguments. Related work is presented and what this project contributes. Also, there is a presentation of future work where there are suggestions about how the project could be extended.

2 Background

Dealing with distributed systems involves several important concepts that are discussed in this chapter. At first, there is a general description of related terms and terminology. The COM technique for building components is described, followed by DCOM which is meant for distributed components, and then COM+ is described. COM+ is an extension of COM and therefore is it natural to describe COM and DCOM before COM+. .NET, a new platform, is then briefly described. Lastly, Enterprise JavaBeans, another technique for building components, is described.

2.1 Distributed systems

Coulouris, Dollimore and Kindberg (2001, p. 1) define a distributed system as "… one in which components located at networked computers communicate and coordinate their actions only by passing messages". This, they say, leads to concurrency of components, lack of global clock and independent failure of components. The main motivation, according to Coulouris et al. (2001), for building distributed systems is resource sharing, where resource means a number of things that can usefully be shared in a distributed network, like printers, files and databases.

Pressman (2000) identifies a number of reasons for why distributed systems have become popular. These are performance, resource sharing and fault tolerance. Performance can be increased by, for example, adding computers to the distributed system. This way is usually simpler and cheaper than for example upgrading a processor on a mainframe computer. Resource sharing means that things like printers, files and so forth, are shared among the computers in the network. Fault tolerance is an important concept in this report, and means that the system tolerates hardware and software faults. In other words, if a number of computers in a distributed system perform the same task, then if one fails another can take over. The concept replication is used in distributed systems to indicate copies of, for example, hardware, data and services. For example, in a replicated database, if one computer containing a data object replica goes down the users can still access the other replicas of that data object.

A common distributed system architecture is the client/server model. The client/server term applies to processes', a process can call another process and asks for its services, then there is a client process, asking for service, and a server process, providing a service. It is not necessary that a server or a client process always plays that role, it is possible that a server, in handling a request from a client, needs to communicate with another process thereby taking the role of a client process.

Another variation of the client/server model is the peer to peer model in which there is no distinction between clients and servers in the system. Instead, all of the processes play similar roles, interacting cooperatively (Coulouris et al., 2001).

2.2 Components, objects and interfaces

The words component and object, when dealing with distributed systems, are often confused, and it is difficult to understand the real difference between them. Below, a general definition of these concepts is provided.

According to Microsoft Corporation (1995) a *component* is a piece of reusable code in binary form that can be combined with other components, either in the same computer or over a network, to form an application. A component runs within a context called a container, which can for example be a web browser. By using components it is possible to divide an application into several smaller pieces, making each piece easier to handle than the entire application.
An *object*, on the other hand, consists of data and code containing functions that represent what the object can do, and associated information for those functions. An object has a well-defined interface, which is a set of related functions, and the functions of the interface are called methods (Microsoft Corporation, 1995). Object often represents something in the real world, for example a person or a car.

Interfaces are used in distributed systems for components to expose their services to other components and objects. Interfaces completely hide implementation details from its user. Coulouris et al. (2001, p. 170) specify interfaces as something that "…provides a definition of the signatures of a set of methods (that is, the types of their arguments, return values and exceptions) without specifying their implementation". Furthermore, they specify that if an object's class contains code that implements the methods of an interface, the object will provide that interface.

2.3 Dependability concepts

Dependability is a collective term for several concepts relating to quality of service in distributed systems. This report uses the definitions given by Laprie (1994).

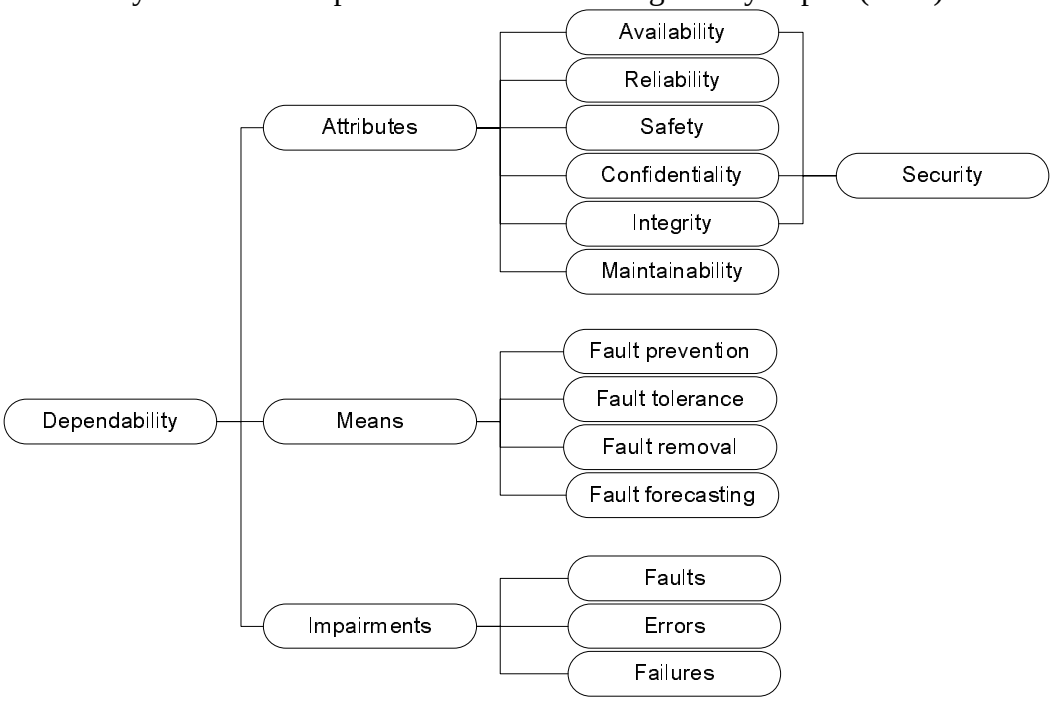


Figure 1 The dependability tree according to Laprie (1994)

Dependability subsumes the attributes of reliability, safety, maintainability and security (Laprie, 1994). The definition of these terms, according to Laprie (1994), is that:

- *availability* is the readiness for usage,
- *reliability* is the continuity of service,
- *safety* is the non-occurrence of catastrophic consequences on the environment,
- *confidentiality* is the non-occurrence of unauthorized disclosure of information,
- *integrity* is the non-occurrence of improper alterations of information,
- *maintainability* is the aptitude to undergo repairs and evaluations.

Security is the association of integrity, availability and confidentiality, with respect to authorized actions. Laprie (1994) continues with the definition of the terms *failure*, *error* and *fault*. A system function is what a system is intended for, and a system failure occurs when the delivered service differences from fulfilling the system function. A part of a system state which is likely to lead to subsequent failure is an error, and an error affecting the service is an indication that a failure occurs or has occurred. A fault is the adjudged or hypothesized cause of an error.

There exists a set of methods, according to Laprie (1994) that the development of a dependable system calls for: fault prevention means preventing fault occurrences or the introduction of faults, fault tolerance is providing a service that fulfills the system function in spite of faults, fault removal means how to reducing the number of faults, and fault forecasting is estimating the present number, the future incidence, and the consequences of faults.

To summarize: availability and reliability emphasize the avoidance of failures, safety the avoidance of a catastrophic failures, and security the prevention of unauthorized access and/or handling of information.

A system is scalable if it remains effective when there is significant increase in the number of resources and users (Coulouris et al., 2001). Adding clients to a distributed system should not require significant changes in the source code and/or resources.

2.4 Middleware

The description of clients and servers above indicates that they communicate with each other directly. This is not entirely true; usually there exists a one layer of software between them called middleware (Pressman, 2000).

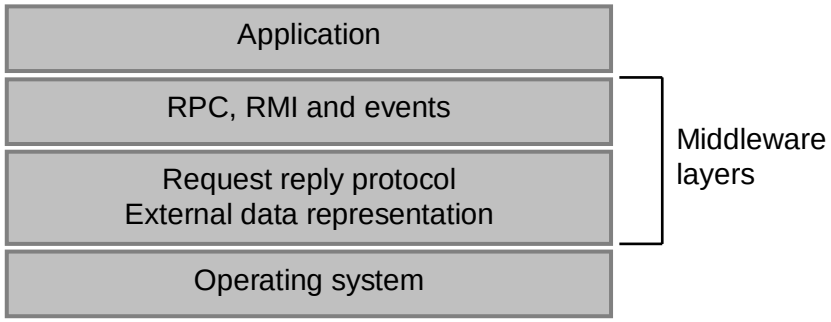


Figure 2 Middleware layers as identified by Coulouris (2001)

The purpose of middleware is to "glue" together different parts of a distributed system. Coulouris et al. (2001, p. 16-17) define middleware as "…a software layer that provides a programming abstraction as well as masking the heterogeneity of the underlying networks, operating systems and programming languages". They define location transparency and independency from the details of communication protocols to be important aspects of middleware. *Location transparency* means that a client calls a procedure and cannot tell whether the procedure is in the same process or in a remote process. In more general, location transparency enables resources to be accessed without knowledge of their location (Coulouris et al., 2001). This is beneficial in distributed systems because it is not necessary to explicitly define the location of the resource that the client needs.

In a distributed system, there needs to be interaction between processes, where one process sends a request to another process on a remote machine and receives a response. This is referred to as a remote procedure call, or RPC. Mullender (1993) describes the structure of RPC. The calling process is a client and the called process is a server. If the client and the server are the same process, the client calls the subroutine directly.
When calling a subroutine in another process the call is made via a *client stub* (Mullender, 1993). This subroutine has exactly the same interface as the server's subroutine, but is implemented to ask the server to execute the remote subroutine and return the resulting value to the client.
The server has a similar piece of code, called a *server stub*. It receives requests from the client stub and calls the actual subroutine, which returns the results. The results are forwarded to the client stub, which in turn returns it to the client. From the client's point of view this looks identical to local procedure call.

Remote method invocation (RMI) is method invocation between objects in different processes either on the same computer or on a remote computer (Coulouris et al., 2001). This works roughly in the same way as RPC; the difference is that RMI is associated with objects.

2.5 Multi-tier architectures

The client/server model described in 2.1 can be extended further. In its simplest form, it is a 2-tier architecture, consisting of only clients and servers. Pressman (2000) says that these kinds of architecture are normally used where little data processing is required. This, he says, causes problems when an application requires considerable processing. Another reason for using a multi-tier architecture is to reduce the coupling between components in the system, making each component easier to implement and handle.

When describing a multi-tier architecture it is common to use the term *layer* instead of tier. In the 2-tier architecture Pressman (2000) identifies the layers to be presentation and logic layer (resident in the client) and a database layer (located in the server). The presentation layer is the GUI, and the logic layer is the program logic, like saving in the database or pressing a button; in other words handling all the data produced by the user and received from the server. Another word for a logic layer is *business layer* or *business logic*, which refers to enterprise systems and the logic that needs to be carried out in order to meet real world requirements. An example of business logic

is a rule that says that doctors can only give a prescription for a drug to a patient if that drug is available at the pharmacy[2].

An example of a three-tiered system is one consisting of a presentation layer (clients), a processing layer (or an application server layer) and a database layer (Pressman, 2000). The purpose of the application server layer is to handle the business logic, receiving requests from the clients, manipulating those requests and giving back a response. The clients are so called *thin clients*, meaning that they mainly present the data to the user, and do not perform any logic themselves. Pressman (2000) describes that this middle tier supports maintainability and reusability and contains objects defined by reusable classes which can be used again and again in other applications. These objects are often referred to as business objects. Pressman describes that these objects contain methods which communicate with the database layer. The presentation layer sends messages to these objects in the middle layer which will either respond directly or carry out a dialogue with the database layer, which would then provide data that would be sent back to the presentation layer (Pressman, 2000).

Figure 3 A three-tier architecture

A typical scenario in a three-tier system is that a client sends a request to the application server, asking it for some service it provides. The application server handles that request and communicates with the persistence storage. On receiving the results from the persistence storage the application server sends these results to the client that displays the results in an appropriate way to the user. In more detail, if working with databases, the application server handles transactions, establishes connections to the database, and so on. The client can be a GUI, a web page or a Windows form, for example.

It is possible to further divide the three-tier system architecture. An example of this is adding another layer called a persistence layer, which makes the architecture 4-tiered. This means that there are the thin clients, a business layer with the business logic, a persistence layer and a persistent storage. Under these circumstances it is possible to place the business layer outside the application server and instead place the persistence layer in the application server. This has some advantages. For example, any one tier can run on a different platform and the tiers can be manipulated and updated independently. Ambler (1999) points out that in order for the user interface to obtain information it must interact with objects in the business layer, which in turn interacts with the persistence layer to obtain objects stored in the persistence storage. According to Ambler, not allowing the user interface to directly access information stored in the persistence storage effectively de-couples the user interface from the persistence layer. This is beneficial since it opens for the possibility to change the way objects are stored, for example by changing tables in the database or exchanging database server without re-coding the user interface.

[2] This is not necessarily a true business rule for Theriak.

With this architecture the business layer handles all business logic but does not handle any persistence logic. The persistence layer instead handles this, so the persistence layer now handles connection to the persistent storage, transactions and so on. Furthermore, since the persistence layer is used for communication with the persistent storage, it is possible for the business layer to simply send an object to the persistence layer and tell it to, for example, save it in the persistent storage, like a database. Since the persistence layer handles all communication with the database, the business layer sends an object to the persistence layer along with a command for what to do with that object, so there is little or no SQL code in the business layer. The persistence layer receives the object along with the command and generates the appropriate SQL code. This makes it possible to exchange persistent storage without changing any code in the business layer (Ambler, 1999).

2.6 A 4-tier architecture

This chapter describes the architecture that is suggested by Theriak and used in this project; it is a distributed 4-tier system that has several layers that can be located on separate machines. This means that the layers must be able to communicate with each other over a network, either an intranet or the Internet. At the top layer are the clients; these clients can be for example PC clients, PDA (Personal Digital Assistant) or web applications. These clients are thin clients, meaning that their only purpose is to display data for the user, providing a graphical user interface for the system. By keeping the clients simple, they are relatively easy to implement and maintain which can be a great benefit in a distributed system.

Just below the clients is the business layer where the business logic resides. The business layer contains collections and collection items, e.g. patients, which are stateful. This means that they contain information about the name of the patient, social security number and so on. Collections are a gathering of collection items. An example of a collection is Patients, which contains a number of Patient collection items. In order to make the objects stateful it is necessary to store their state between invocations of the application. This is done by storing their state in the database. If the business layer wants to manipulate an object, like saving, restoring or updating it, it sends the object to the persistence layer.

The data (data here is the object hierarchy of collections and collection items) between the business layer and the persistence layer is encoded in XML format. XML is a markup language that provides a way to share data. It allows own definitions of tags, for example can Patient be a tag that contains other tags like name and address.

The business layer packs object or object hierarchy into XML format before sending it to the persistence layer, which needs to unpack this XML data. Then the persistence layer interacts with the database, and sends the results back to the business layer after converting them back to XML format. So both the business layer and the application server need to have the ability to pack/unpack data to and from the XML format. The data stored in the database are records of drugs, patients, etc.

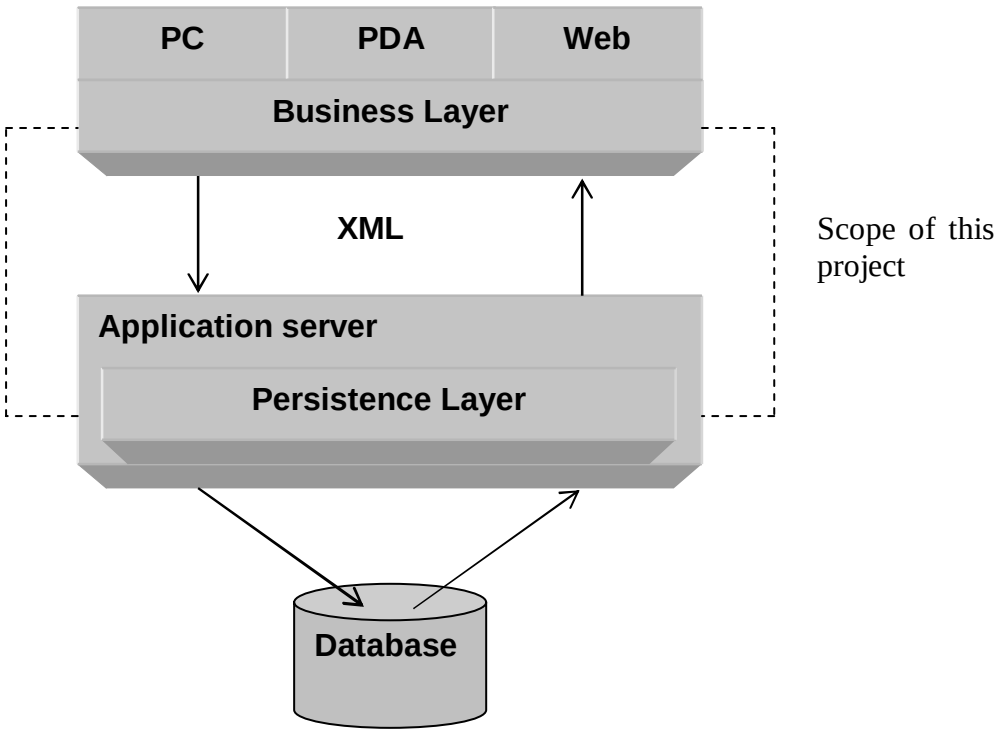


Figure 4 The system architecture suggested by Theriak

Figure 4 shows the architecture that is considered in this project. The persistence layer is responsible for database communication and the manipulation of persistent objects (Ambler, 1999). This involves transaction support, saving and restoring data etc. The persistence layer is stateless; it does not have to know the state of the object which it is manipulating. The reasons Theriak gives for separating the business logic from the persistence logic is to reduce the number of concurrent connections to the database, make the installation of clients simpler and to make the system more modular. The reason for using XML for encoding objects is to reduce the roundtrips between the business layer and the persistence layer. The business layer can send an object hierarchy encoded as XML to the persistence layer, which responds with the results, instead of making the business layer send each object separately and receive a separate result for each object it sends. This might occur when the user is working with more then one patient at the same time, storing them all at once when for example pressing a save button in the GUI.

2.7 COM technologies

According to Microsoft Corporation (2002a), COM is a standard; it is a platform-independent, distributed, object-oriented system for creating binary software components that can interact with each other. The keyword here is binary standard; COM components can be written in any programming language (it is a requirement that the programming language is capable of handling pointers), like Visual C++, Delphi and Visual Basic (Freeze, 2000). Freeze describes COM as a technology that combines the ability to create individual components with the ability to create reusable objects. This, he suggests, gives the possibility to create binary object modules that are independent of any programming language and with well-defined object-oriented interfaces.

2.7.1 COM interfaces

Microsoft Corporation (1995) defines interfaces as the way applications interact with each other and the system through a collection of functions calls (also known as methods, member functions or requests). This interface is a strongly typed contract between software components, and provides a useful set of semantically related operations.

Microsoft Corporation (1995) also states that COM is a technology that allows objects to interact across process and machine boundaries as within a single process. This is possible since the only way to manipulate data associated with an object is through its interface. A COM interface does not necessarily represent all the functions that a COM class supports. When an object "implements an interface", the object implements each method of the interface and provides a COM binary-compatible pointer to those functions to COM. These functions are then available for any clients who ask for a pointer to the interface, whether the client is inside or outside the process that implements those functions.

According to Microsoft Corporation (1995), a client calling a COM object never has direct access to the object in its entirety, which is different from for example C++ objects. Instead, the client always has to access the object through clearly defined contracts or the interfaces that the object supports and only those interfaces. Furthermore, a COM interface only defines how one would use that interface and what behavior is expected when accessing object through the interface; interfaces do not define any implementation details whatsoever

There exists a convenient standard to represent object and their interfaces in figures. The standard is to draw each interface of an object as a "plug-jack", as illustrated in Figure 5.

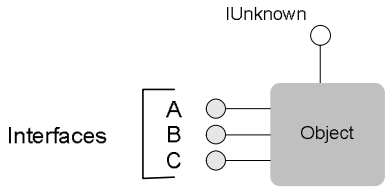


Figure 5 A standard way to draw interfaces (Microsoft Corporation, 1995)

The idea behind this "plug-in jack" concept is that the client must have the right kind of plug to fit into the interface jack in order to do anything. In this case the client must have an A plug to use the services provided by the object. If the client has for example a D plug it would not be able to use this object. This can be compared with having a stereo system that has a number of different jacks for input and output, like a headphone plug in. Then you could plug your headphones or your loudspeakers to the stereo system but it would be harder to plug in your washing machine.

2.7.2 COM clients and servers

Continuing with Microsoft Corporation (1995), a critical aspect of COM is the interaction between clients and servers. A COM client is a piece of code or an object that receives a pointer to a COM server and uses its services by calling the methods of its interface. A COM server is any object that provides services to clients in the form of COM interface implementation.

There exist two different kinds of server, one is the *in-process* server whose code is executed in the same address space as the client, and the other is the *out-of-process* server which runs in another address space or even in another machine. The two different flavors of out-of-process server are sometimes called *local server* and *remote server*.

COM supports location transparency. From the client's point of view this means that all objects are accessed through interface pointers. The pointer must be in-process, but any call to an object function always reaches some piece of in-process code first. This is because if the object is in-process the call reaches it directly. Otherwise, if the object is out-of-process, then the call first reaches what is called a *proxy* object that is provided by COM itself and this proxy then generates the appropriate RPC to the other process. The server, on the other hand, has a different view. Since calls are made using interface pointers, which only have context in a single process, the caller always has to have some in-process code. If the object is in-process the caller is the client itself, otherwise the caller is a *stub* object provided by COM that picks the RPC from the proxy in the client process and turns it into an interface call to the server object. In other words, clients and servers always communicate directly with some other in-process code. This is illustrated in Figure 6, which is based on figure in the COM specification (Microsoft Corporation, 1995).

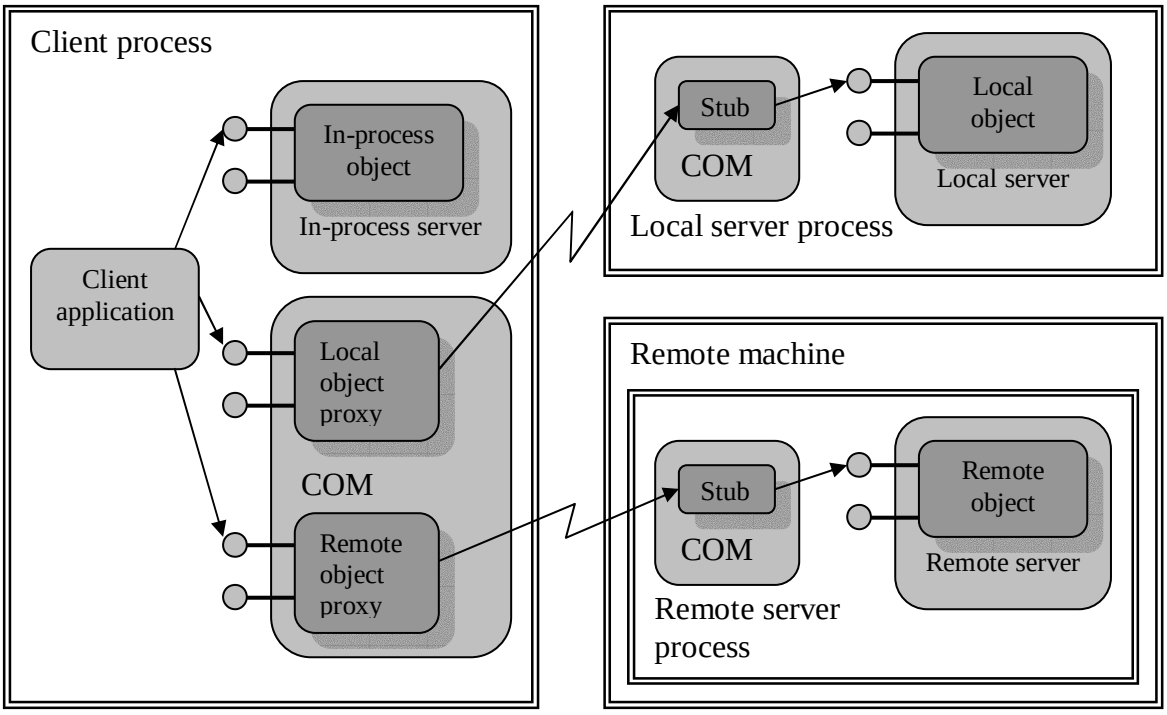


Figure 6 Client server communication in COM

To summarize, dealing with objects that are in the same process and those that are in another process seems identical to the application programmer. These issues are further discussed in the next chapter about DCOM.

2.7.3 DCOM

Freeze (2000) defines DCOM as the combination of COM with additional network protocols that allows running COM objects on a remote computer. This is similar to out-of-process COM object, except the application does not know the machine on which the object runs; instead that information is kept in the Windows Registry. This means that the application never really realizes that the COM object is located on another machine since the COM object works as if it were on the same computer, except for any delays caused by the communication over the network. The application looks for the object in the local machine, just as it normally would, but instead of finding the information about the object on the local machine it finds a reference to a remote machine were the object is located. This is the biggest advantage of using DCOM over running COM, i.e. the program doesn't need to know where the object is located. This information can be set and changed in the Windows Registry as needed without affecting the application (Freeze, 2000).

This means that the only difference between COM and DCOM is that the application programmer needs to specify the name of the remote computer in COM, while DCOM automatically locates the COM object (Freeze, 2000).

The location of a component is completely hidden in DCOM, whether the component is in the same process or half way around the globe. The way the client connects to and calls a component is identical in all cases. In other words, DCOM does not require any changes in the source code; it is not necessary to recompile anything (Freeze, 2000). A simple configuration using a tool called DCOM configuration tool is enough to change the way components connect to each other. This greatly simplifies the task of distributed application components.

2.8 COM+

COM+ is an extension to COM, and it is possible to use COM objects in COM+. As COM was developed as a standard for building object-oriented programs, COM+ was developed as a standard for designing distributed, multi-tier applications
In COM+ it is possible to move and copy components, which means that it is possible to configure a single physical implementation of a component many times. This configuration is done at the binary level rather then the source code level, which results in less code (Microsoft Corporation, 2002b).

Löwy (2001) describes services that are provided by COM+:

- Administration, which is a set of tools for developers and administrators to configure and manage components and component-based applications.
- Just-in-Time Activation (JITA) are services that instantiate components when they are needed and discard them when their work is done.
- Object Pooling are services that allow instances of frequently used resources to be maintained in a pool for use by several clients.

- Transactions, which are services that are carried out by distributed components, and resources are allowed to be manipulated as a single operation by these services.
- Synchronizations are services for controlling concurrent access to objects.
- Security is used to authenticate clients and controls access to an application, COM+ supports role-based security. Role-based security means that users are assigned to roles and access rights are given to those roles, in contrast of defining access rights directly to users.

According to Löwy (2001) COM+ services (or just COM+) are the services that support COM and .NET component based applications.

2.9 Enterprise JavaBeans

Sun Microsystems (2001, p. 1) definition of EJB 2.0[3] is the following:

> The Enterprise JavaBeans architecture is a component architecture for the development and deployment of component-based distributed business applications. Applications written using the Enterprise JavaBeans architecture are scalable, transactional, and multi-user secure. These applications may be written once, and then deployed on any server platform that supports the Enterprise JavaBeans specification.

There are three different kinds of EJB; *entity beans*, *session beans* and *message-driven* beans. Monson-Haefel (2001) points out that a good rule of thumb is that entity beans model business objects that can be expressed as nouns, like patient, and are often persistent records in a database. Session beans on the other hand are more like verbs, they are responsible for managing processes and tasks. This can be for example to register a patient's arrival to a hospital; this might call for using a Hospital bean, but there is also a need for various information about the patient itself, like name and symptom. Session beans have a lot to do with the relationship between different enterprise beans, and tend to manage activities like the act of registering a patient's arrival. This also means that session beans do not represent something in a database. Message-driven beans have the responsibility to coordinate tasks involving other session and entity beans. A message-driven bean listens for specific asynchronous messages to which it responds by processing the message and managing the actions that other beans take in response to those messages.

[3] If not stated otherwise, in this report EJB refers to version 2.0

2.9.1 EJB interfaces and classes

EJB has a number of components: remote interface, remote home interface, local interface, local home interface and two classes: bean class and primary key. In order to implement a bean it needs interfaces and one or both classes. Monson-Haefel (2001) defines this further;

- A remote interface defines the bean's business methods.
- A home interface defines the bean life cycle methods; this includes creating, removing and finding beans.
- A local interface defines the business methods that can be used by other beans that co-exist in the same address space, the same container. This makes it possible for the beans to interact without the overhead of distributed objects.
- A local home interface defines the bean life cycle methods that can by used by other beans co-existing in the same container.
- A bean class is the class that actually implements the bean's business methods and life cycle, but not the bean's component interface. The class must have a signature matching the remote interface methods and must have methods corresponding to some of the methods in the home interface. Since the message-driven bean is never accessed by methods calls from other applications and beans, it does not use the component interface. It only needs the bean class to operate.
- A primary key provides a pointer into a database. Only entity beans need this class.

It is possible to implement a bean that only has a remote interface, if it can be guaranteed that it will only interact with remote clients. The same goes for local interface, if a bean is only to be used within its container it is not necessary to implement remote interface for it.

Perrone and Chaganti (2000) further divide session beans into stateless and stateful session beans. Stateless session beans are those that are created with no regard to subsequent calls by a client. Stateful session beans are those that maintain a state for a particular client between subsequent calls, this is before some maximum amount of time has expired.

Perrone and Chaganti define two types of entity beans; Bean-Managed Persistence (BMP)-, and Container-Managed Persistence (CMP) entity beans. BMP are those where all code dealing with database operations (to a relational source) are done by the EJB developer. CMP are those where these operations are performed by the EJB container implementations.

2.9.2 EJB container

Monson-Haefel (2001) says that there are many interactions between a bean and a server, and a container is responsible for presenting a uniform interface between the bean and the server. In other words, the container is responsible for creating a bean, making sure it is stored properly by the server, and dealing with security, transactions, naming and other services common to distributed objects. *Deployment descriptors* are used for the EJB container to know how to apply these services to each bean at runtime. The deployment descriptors allow setting the behavior of enterprise beans at

runtime without changing the beans themselves. In other words, with deployment descriptors it is possible to customize the runtime attributes like security, transactions etc. This can be done with visual tools, and is done without altering the bean's interface or class.

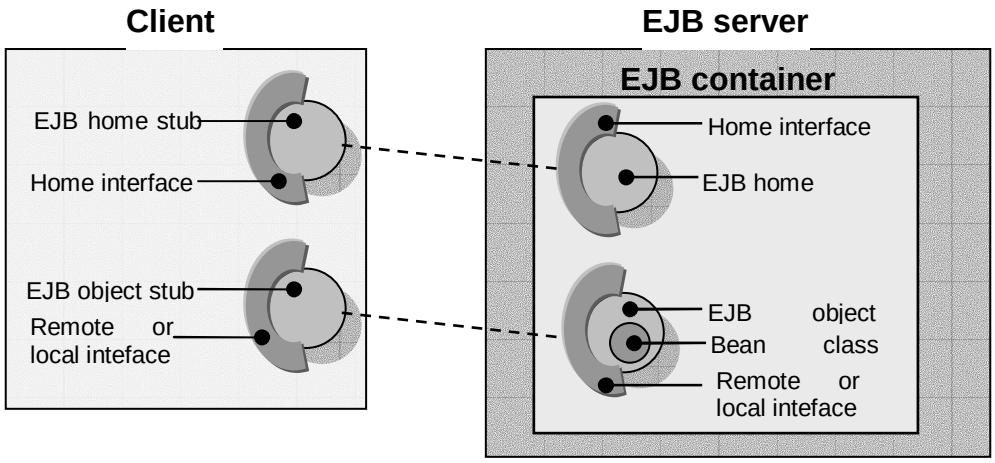


Figure 7 EJB architecture as in Monson-Haefel (2001)

Both entity beans and session beans define remote interfaces that clients outside their container interact with; and local interfaces for clients that are in the same container. Monson-Haefel (2001) discusses this in more detail, by talking about EJB objects and EJB homes. The EJB object is an object that implements the remote and/or local interfaces of an enterprise bean. EJB home is similar to EJB object; it is another class that is generated automatically when the bean is installed in a container. It implements the objects methods and is responsible for the bean's life cycle. Both the EJB object and the EJB home are generated automatically during the deployment process.

3 Problem definition

Whether to use COM+ or EJB to implement a distributed system, considering the criteria needed to fulfill the problem, can be difficult for application developers (Sessions, 2000a). Sessions (2000a) further describes that COM+ is limited to the Windows platform, and Enterprise JavaBeans are dependent on the Java programming language. There are also other issues that need to be considered; depending on the type of industry that the application is intended for, different criteria are not necessarily equally important. For example, dealing with information systems for healthcare organizations, dependability aspects are important. Since healthcare organization deal with sensitive data security is paramount. Reliability and availability issues are significant, considering that lives may depend on the system working correctly in stressful situations. Another example is a system where throughput or timeliness might be of more importance, such as a control system in an airplane.

COM+ and EJB are two different techniques to build component-based applications, and they handle different criteria in different ways. Discovering in the implementation phase that either technique does not support criteria efficient enough is undesirable. The aim of this project is to provide a recommendation about which technique to use to build a multi-tiered distributed system given certain criteria. In order to achieve this aim the following objectives should be met. The first objective is to study which mechanisms in COM+ and EJB support scalability and Laprie's (1994) dependability aspects, with focus on security, reliability and availability considering the architecture described in 2.6. These are also criteria that can be considered to be important in many systems today (Pressman, 2000). Second objective is to discuss how these techniques can be used in a persistence layer in a multi-tiered architecture. Third objective is to compare the two technologies and based upon this comparison provide a recommendation. Furthermore, this report will apply this evaluation to Theriak, a company dealing with products for the advancement of healthcare organizations, using the company's system as a case study.

3.1 Evaluation criteria

One of the purposes of this report is to identify the options and functionality that the COM+ and EJB techniques provide. Advantages and disadvantages of these techniques are discussed, focusing on the persistence layer. The evaluation criteria are scalability and dependability, with focus on security, reliability and availability. These concepts are defined in section 2.3 and the architecture is described in section 2.6.

Medical data, like information about patients, can always be regarded as sensitive. Therefore, the security aspects are important. A number of security threats can be identified, like it should not be possible for someone to get access to the application server and thereby get access to the database and all the data. Therefore it is necessary for the clients to be able to authenticate themselves to the application server in order to get access to the database, and the application server should be able to handle different roles so that the role the client plays decides what data it can manipulate. Also, the server needs to authenticate himself to the clients, so that a client can know that it is communicating with the right server.

When building a large enterprise system, it is important that the system is scalable. This means that it should not be difficult to add a number of clients to the system, or to add servers to the system. A number of solutions exist to increase the scalability of a system. For example, load balancing is important, which means that a server should be able to handle requests at peak times, and if it does not handle all requests it should redirect the traffic to another server or handle the requests at a later time, if possible.

Dependability is crucial for information systems in medical institutions. When dealing with medical data, like information about patients, drugs and etc. it would be very serious if the system would suddenly crash and leave everything in an inconsistent state. For example, if a nurse updates data denoting that a patient has been given the daily dose of a certain drug and the system fails, another nurse might consider that this patient has not been given its daily dose and give him another dose, which might have serious consequences.

3.2 Delineations

This section discusses some aspects that are not covered in this report and the reason why. First of all, details about the relation between the application server and the database server are not covered. Although it is necessary to mention some aspects that have to do with persistent objects and a database, it will be in a more general way rather than details. The same goes for the clients, they are not considered in detail since they are not directly relevant to the problem. Although there is some implementation involved in this report, there is no kind of prototype for the system. The purpose of the code examples is to strengthen the arguments for the technique that is recommended.

3.3 Expected results

The goal of this study is to point out and discuss the options that COM+ and EJB provide to Theriak. For example, considering security issues there are several ways to implement and manipulate security in both COM+ and EJB, but what is important to Theriak is which technique has more to offer or is considered more appropriate for application programmers. For example, one technique could be simpler, more understandable, and so on. With this in mind, the result is not that “COM+ is better than EJB” or vice versa, the result is a recommendation about what technique is considered to be more appropriate for Theriak to solve its problem. Alongside this recommendation will be discussions about the technique and arguments why it is recommended over the other technique.

4 Method

This chapter describes the methods that can be used to solve the problem defined in chapter 3. This chapter evaluates different approaches for solving the problem and describes the chosen approach.

4.1 Approaches

There are several possible approaches to solve the problem. One approach is to conduct a survey. This involves inspecting real world examples of 4-tiered distributed systems that are implemented with COM+ and EJB, and interviewing people that have designed and implemented the system. The major problem with this approach is that it is unrealistic to find two equal 4-tier systems, one implemented with COM+ and the other with EJB. Inspecting only one system that is implemented in either technique would affect the results negatively since the analysis would not be objective.

Another approach is to design and implement prototypes in both COM+ and EJB. The prototypes could be used to make measurements, i.e. design test cases and analyze the results from those tests. This approach, however, requires more time than is available for this project.

The third approach is to theoretically evaluate criteria. The criteria are specified in advance and are decided on the basis of important factors in a multi-tiered distributed system. The technologies are studied in relation with the criteria. This approach demonstrates how the technologies handle the different criteria. The technologies are compared and the results are based both on the analysis of the studies and the comparison of the technologies. This is the approach used in this project.

In the previous chapter some objectives were identified in order to accomplish the aim. In order to achieve the first objective, study how the techniques support the criteria, literature study is used. For the second objective, applying the technologies to a persistence layer, we discuss how the different technologies can be used to support implementation of a real-world architecture. The third objective is achieved by doing a side-by-side comparison of the technologies which results in a recommendation based on the criteria, satisfying the aim of the project.

4.2 Literature

The main books used in this report are of different flavors, but all have been published in the last two years. The contents of these books vary. Some guide the reader through developing components and cover most of the common features of the technology. These are books that provide programming examples of components, in a programming language like Visual Basic or Visual C++ for COM+, and naturally Java for EJB. Other books do not provide programming examples for components but rather smaller examples to illustrate the technique.

There exist few scientific papers on the use of component technology for multi-tier systems. The articles that do exist are often bound to specific environmental issues and problem areas, which limit their usefulness for a general-purpose study. Some articles are published by the creators of these technologies, i.e. Microsoft and Sun Microsystems, and are therefore not considered appropriate here. This is because these are often used to promote a specific technology, and can therefore not be considered objective studies. This type of papers can, however, be useful as a source for technical descriptions.

With the previously described literature, both techniques are considered independently from one another. This is done to illustrate how COM+ and EJB handle the scalability and dependability attributes.
Pointing out what options COM+ and EJB provide for the application programmer gives an idea of how the respective techniques handle different criteria. Options here mean what the technique provides for the application programmer, i.e. the support it has for the criteria.

4.3 Programming language considerations

Besides the literature survey, small programming examples are used to show how COM+ and EJB use different approaches for different problems, and examples of components is provided in Appendix A: An example of a COM+ component and Appendix B: An example of a session bean. Since EJB is Java based, the Java programming language will be used to show the programming examples in EJB. In fact, Java is the only programming language that can be used to implement enterprise beans. COM+, on the other hand, supports more programming languages, but for ease of reading Visual Basic is used. This is because even though the programming examples are meant to provide further understanding of the technique, the intention of this report is to not go too much into programming details, which would probably be the case if for example Visual C++ would be used. A possibility would be to also use Java in COM+ programming examples, since COM+ itself is language neutral, but according to Sessions (2002b) Sun's Java is particularly difficult to use for implementing COM+ components.

4.4 Chosen method

The method consists of a literature survey on COM+ and EJB. This study is used to evaluate how these techniques handle the different criteria listed in section 3.1. At first, the relevant literature for the problem is studied. This is done to comprehend the necessary terminology used in COM+ and EJB. Additionally, this literature is the backbone for this report and consists of text books and articles.

This report compares two different approaches to solve a problem, and in the light of this comparison, and to achieve the first objective, it starts with looking at each approach independently. The techniques are considered in relation with the attributes scalability, security, scalability, availability and reliability.

After these initial studies, there is a discussion about how the techniques can be used in the persistence layer, which is the second objective. In this discussion the facts that were gathered in the first objective are used. This objective uses a real-world architecture discussed in section 2.6. Even though this architecture is meant to be used in a healthcare organization environment, it can even be used in other enterprise

environments. This architecture is used in this report to demonstrate how the techniques can be used in a real-world environment.

The actual comparison, the third objective, discusses and evaluates the studies of the techniques, their differences and similarities. This objective is important when considering the aim, to provide a recommendation about which techniques to use in a multi-tiered distributed system given certain criteria. This way, it is possible to refer to the previous chapters for further details about the techniques.

5 COM+

COM+ is Microsoft's component technology and traces its roots back to 1993 when COM was first introduced. Today it plays a large role in the Windows 2000 and Windows XP platforms. The *Microsoft Management Console*, or MMC, is the application programmer's view of the COM+ environment (Sessions, 2000b). MMC allows for the administration of many of the services of a computer running a Windows operating system. With MMC the COM+ components are introduced to the COM+ runtime environment, and its characteristics (like security) are defined.

5.1 Scalability

Sessions (2000b) describes component instantiation, which is the creation of the proxy (in the client process) and the stub (in the server process) for a component. This instantiation can be *load balancing* which means that the instantiation request from the client goes through an intermediary machine called the *Component Load Balancing* (CLB) Router.

5.1.1 Component Load Balancing

The CLB Server chooses an application server to create the stub, and thereby creates the instance, so indirectly CLB Server is choosing the application server on which the instance will run. This is demonstrated in the following figure.

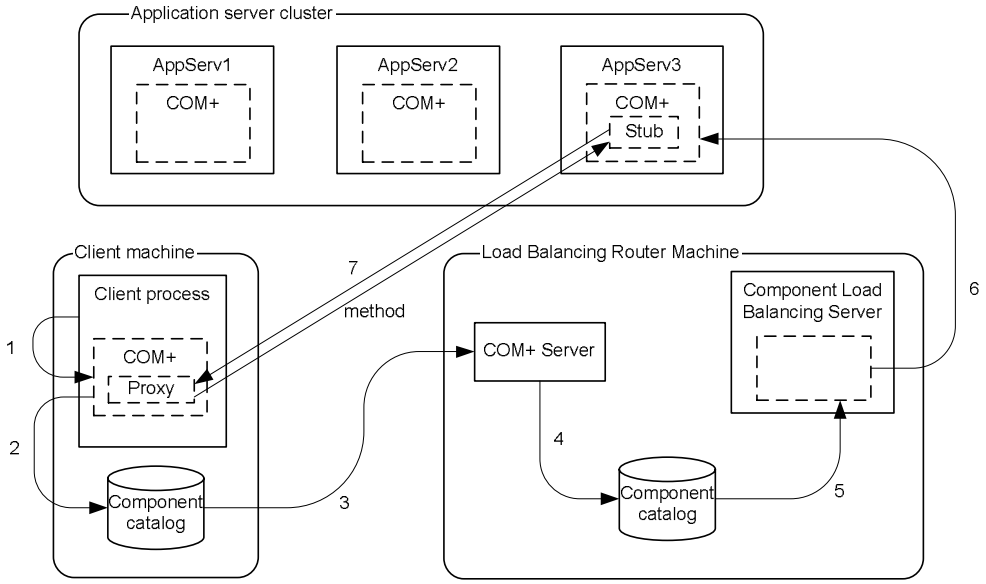


Figure 8 Load balancing (Sessions, 2000b)

What is happening in Figure 8 is that the client makes an instantiation request (1) and the local COM+ system checks with the local component catalog where the component resides (2). The component catalog believes that the component resides in the COM+ router machine that hosts the CLB Server (3). The COM+ runtime environment looks up the component in its catalog (4) and discovers that the component is load balanced (5). The CLB Server forwards the instantiation request to an application server in the cluster (6), in this case AppServ3, assuming that the CLB Server decides that it is most available. AppServ3 creates a local stub and through the COM+ runtime environment returns a proxy to that stub back to the original client

(7). This means that all subsequent method invocations via that proxy go directly to the stub (Sessions, 2000b).

5.1.2 The object-per-client model

It is possible to let the application servers each have their own COM+ object representing for example a patient. All clients accessing this patient through the same application server use the same physical identity, sharing the same object. This means that their calls would need to be handled concurrently, which means multiple threads (Ewald, 2001). This puts a burden on the component developers since they now have to deal with concurrency. In COM+ this can be avoided by using the *object-per-client model* described by Ewald (2001). Using multiple COM+ objects, scalability is enabled by providing a way to distribute work among multiple server machines.

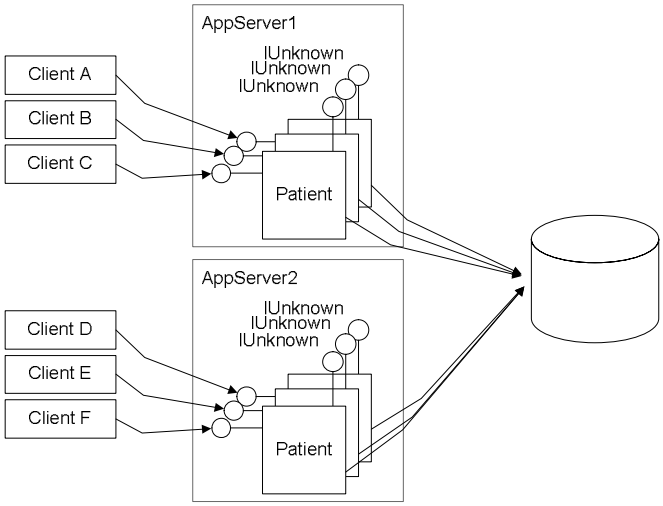


Figure 9 The object-per-client model (Ewald, 2001)

Since the architecture has stateless application servers there is no need to be concerned about access to any shared state in the objects. By using the object-per-client model it is possible to know how many clients are using the system, since each has its own object. This can be useful information because it provides a way to track and possibly limit concurrency on a particular server (Ewald, 2001). Another benefit of using this model is that it is relatively easy to know which client made which invocation.

In the architecture described in section 2.6, the object-per-client model could be used to achieve scalability. The objects are stateless which means that clients can use any available object to serve their requests. The CLB can get information about how many clients each application server is serving by getting the number of objects.

5.2 Dependability: Security

In COM+ it is possible to set different *levels of authentication*, depending on how secure the information sent between the client and the server needs to be. This authentication level has to be correctly used because setting too high authentication level decreases the performance, because the higher the security settings are the more security checks the server needs to perform before allowing/denying the client access to a service, and setting it to low opens up security holes (Sessions, 2000b). The

levels of authentication according to Mojica (2001) are: none, connect, call, packet, packet integrity and packet privacy.

None is the lowest level of authentication and means that there is no authentication at all.

Connect means that the server verifies the credentials of the client; the parameters are not protected in any way. Credentials are used by principals to prove their identity; a principal can either be a person or a computer. This means that this level of authentication prevents someone without credentials to access the server and this verification is only done when the connection is first established.

Call upgrades the client authorization check (the connect level) to every single method request. This means that instead of the verification only being performed when the connection is first established, it is performed in every method request.

Packet authentication level says that the header for each packet is signed with a session key. This is the default level of authentication. If someone tampers with the message the header changes and is rejected by the server.

Packet integrity means that even the parameters sent with the method invocation are encrypted with the session key.

Packet privacy means that all the information sent is encrypted with the session key. This means that COM+ authenticates the client with every method invocation, encrypts the method, parameters and client identity. This is the highest level of security.

Beside the authentication level there is another security characteristic called *impersonation*. Impersonation is the component process's ability to take on the identity of the client (Sessions, 2000b). Impersonation is necessary if the database manages the security of data rather than the COM+ runtime environment; this is not recommended by Sessions, and not the approach in this project. Here, the security is handled in the middleware.

It is possible to manage all these security characteristics either declaratively through the Component Services MMC or programmatically.

5.2.1 Declarative security

Declarative security is managed in the Component Services MMC; in the MMC it is possible to set the level of authentication discussed in the previous section as seen in Figure 10.

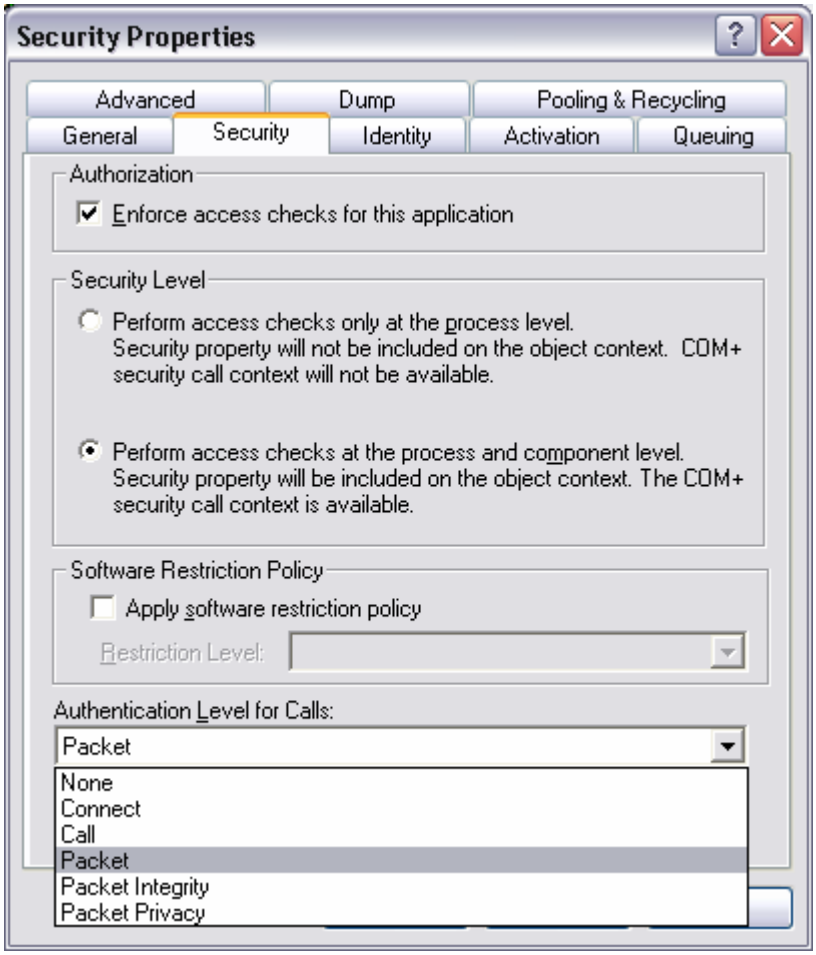


Figure 10 Security settings in MMC

With declarative security it is possible to assign a given role to the entire application, to a particular component, to a particular interface or to a particular method. Role-based security is a declarative security.

The definition of roles makes sense only in the context of COM+ and only on the server machine where the role is created (Mojica, 2001). In other words, the roles are defined in the Component Services MMC and then it is possible to add Windows user accounts to that role.

As can be seen from Figure 10 there are two options for security level. The former, the application security, controls who are allowed to launch the application and instantiate components. The latter, the process- and component security level, also controls who can access each method (Mojica, 2001). This is the choice that a company like Theriak would need. For example, different roles, like a doctor and a nurse, would not have identical access to methods. Some methods, for example, only doctors are allowed to call.

5.2.2 Programmatic security

In Visual Basic the main interface related to security is `SecurityCallContext` (Mojica, 2001). The call context means that the interface gathers information about the causality. In other words, it keeps track of the call chain, i.e. which clients have been involved in the call. COM+ keeps track of users that were involved in each method call and what group they belong to. To obtain this information a function called `GetSecurityCallContext` is used, which returns a pointer to the `SecurityCallContext` interface. The `SecurityCallContext` interface has three methods, `IsCallerInRole`, `IsSecurityEnabled` and `IsUserInRole`.

An example of how to use this method is the following pseudo code, where a doctor assigns a medicine to a patient. This example begins with checking if the security is enabled. If the security is enabled it checks whether the caller is in the role of "Doctor", since only doctors are allowed to access this method. If the caller is a doctor it is allowed to assign a medicine to a patient, otherwise an error is generated which basically says that only doctors are allowed to perform this actions.

```
Public Sub AssignMedicine(Info about patient and medicine)
   If GetSecurityCallContext.IsSecurityEnabled = True Then
      If GetObjectContext.IsCallerInRole("Doctor") Then
         'Assign medicine to patient
      Else
         'Generate error
      End If
   End If
End Sub
```

It is possible to get information about the last caller, the original caller and the total number of callers.

However, using programmatic security means that roles are declared in the source code, which means that they are more difficult to manage. Declarative security is therefore a much better approach since it is relatively easy to add/remove roles and add/remove users that belong to a role. If a nurse would some day become a doctor, it would just be a matter of changing the role in the Component Services MMC. Some critical situation in a system like Theriak's might someplace need to use programmatic security, although it should be avoided.

5.3 Dependability: Availability and reliability

This chapter covers how availability and reliability are supported in COM+. By using clusters of application servers it is possible to increase availability. Defensive programming and error management can be used to increase reliability.

5.3.1 Clustering

Figure 8 describes the load-balancing architecture of COM+ where there is an application server cluster. This figure applies to availability too, and the object-per-client model described in section 5.1.2. For high availability the incoming client requests must be re-routed and distributed evenly among the application servers in the cluster.

Today clustering is managed in the Windows 2000 Advanced Server and Datacenter Server operating systems (Microsoft Corporation, 2002c). There are two clustering technologies that can be used either separately or in combination with each other; these are called *Cluster Service* and *Network Load Balancing* (NLB). The cluster service is mainly intended to provide failover support for database applications. The Network Load Balancing load balances incoming IP traffic across clusters; this is for front-end applications such as web pages.

The Cluster Service enables joining several servers and databases into a single unit. It also provides application interfaces and tools to create cluster-aware applications (Windows Corporation, 2002c). Individual servers in a cluster are often referred to as nodes. A *Cluster service* refers to a collection of components on each node that perform cluster-specific services. This is not to be confused with COM+ components; the components here are various services, for example a so-called *Failover Manager*. *Resource* refers to the hardware and software components that are managed by the cluster service. *Resource groups* are logical collections of cluster resources.

The failover manager is responsible for stopping and starting resources and initiating failover of resource groups. If a resource fails, the failover manager either restarts the resource or takes it offline. The latter case indicates that the ownership of the resource should be moved to another node and restarted under the ownership of the new node (Microsoft Corporation, 2002c). In other words, when an entire node in the cluster fails, its resource group is moved to another available server in the cluster. When a node comes back online, the failover manager can move some resource groups back to the recovered node.

This feature is very important to companies like Theriak which deal with healthcare organizations where critical data is handled. Providing such a failover increases the availability of the whole system since if one server goes down, another one takes over. Since the architecture describes stateless application servers no state is lost if a server fails; the client could be notified of the failure, causing it to try to reconnect. Then it would be routed to another available server.

5.3.2 Error handling

In order to achieve reliability, defensive programming can be used. This means catching errors before they become a failure, and handling these errors. Brown, Baron and Chadwick (2001) describe two architectures of the COM+ error handling model; result code and context information.

Every COM+ interface must return a result code of type HRESULT for each nonlocal method in the interface (Brown, et al., 2001). This provides status reporting to the COM+ middleware. For example, an object can return an error code indicating that a remote host is no longer available. Result codes must be defined relative to the interfaces from which they can be returned, rather than relative to the interface implementation. In other words, when a public interface is defined it is necessary to provide an exhaustive list of all result codes that can be returned from any method. This does not apply to system errors, however, since they can be returned whether or not they have been defined for the method in question. It is possible to define errors of one interface in terms of those for another interface. This means that an interface, like `IPatient` might return results codes in addition to those that can be returned

from another interface like `IHospital`. This requires that the application developers consider result code value collision; a collision between the result codes of `IPatient` and `IHospital` must be avoided since it could prevent the caller from determining which condition has actually occurred. An implementation of `IPatient` can propagate all errors it receives from `IHospital` to its caller without further inspection.

An object implementation can deliver additional context information about a resulting error code that is returned (Brown, et al., 2001). This information can include a human readable description of the error, the GUID of the interface that defined the error, a path of a help file that describes the error, a help context identifier within the help file, and the id of the class that generated the error.

Depending on the programming language, the error management can vary significantly (Brown et. al., 2001). In Visual Basic most of the error management is implemented and used by the Visual Basic runtime and is therefore invisible to the programmer. Catching COM+ errors is equivalent to catching errors raised by Sub procedures, functions or properties. The `on error` command is used to delegate to a handler code within the current procedure, suppress the error, or propagate it to the caller. Then handler code inspects the so called `Err` object to determine the error's properties.

6 EJB

Enterprise JavaBeans is a component based technology from Sun Microsystems. Monson-Haefel (2001) identifies *primary services* that are needed to complete the Enterprise JavaBeans platform. These include: concurrency, transactions, persistence, distributed objects, asynchronous messaging, naming and security. EJB servers automatically manage these primary services, thus making them transparent to the application developer.

6.1 Scalability

Stateless session beans do not require the conservation of state within the EJB that is specific to a client, as described in 2.9.1. They are not saved in the database since they do not contain any data and are not dedicated to one client. Using stateless session beans, according to Perrone and Chaganti (2000), allows EJB container flexibility in maximizing the efficiency in maintaining such EJBs. This is because any instance created by the container can be used by any client at any time. This means that the container can maintain a pool of instances that are allocated to clients as needed regardless of which instance belongs to which client (Perrone & Chaganti, 2000).

6.1.1 Type of enterprise beans

Scalability is increased as the container can create and destroy bean instances as needed. Sessions beans help reduce the network traffic, according to Monson-Haefel (2001). This is done, according to him, by limiting the number of requests needed to perform a task. Using RMI to communicate between objects requires the data to be streamed between the stub and the skeleton with every method invocation. With session beans, however, this interaction is kept on the server. In other words, when a client invokes a session bean, that bean might invoke other methods on other method beans, and then return a result. The network sees only the traffic produced by one method call, for the cost of one method invocation the client gets several method invocations (Monson-Haefel, 2001).

This is illustrated in Figure 11.

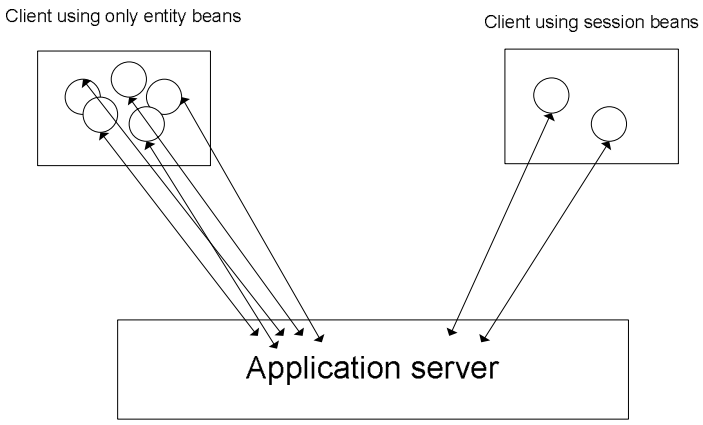


Figure 11 Session beans and network traffic, as shown by Monson-Haefel (2001).

As can be seen from Figure 11, which compares a client using entity beans only and a client using session beans, session beans can reduce the needed network traffic which improves the performance of the whole system. Furthermore, session beans reduce the number of stubs used by the client, which saves the client memory and processing (Monson-Haefel, 2001).

Stateful session beans maintain conversational state between invocations, which means that they are not written to the database but their state is kept in memory while a client uses a session (Monson-Haefel, 2001). Each time such a bean is invoked its state might change, and that change can affect subsequent invocations. This state is kept as long as the client application is actively using the bean (Monson-Haefel, 2001). Stateful session beans are not shared among clients and are therefore dedicated to a single client. However, in the architecture the persistence layer is defined to be stateless which means that it will not contain stateful session beans.

6.1.2 Stateless session beans

This chapter further describes the stateless session beans; in the architecture description in section 2.6 the application server was defined to be stateless, which means that it consists of stateless session beans.

Figure 12 shows an UML (Fowler & Scott, 2000) diagram over a basic session bean (Perrone & Chaganti, 2000). `Patient` is the session bean and it must implement the `SessionBean` interface. The `setSessionContext` method defined on a stateless session bean is used to pass an interface of `SessionContext` to the EJB (Perrone & Chaganti, 2000). The `SessionContext` interface provides access to the runtime session context that the container provides for the session beans instance, and it is passed after the instance has been created (Sun Microsystems, 2002).

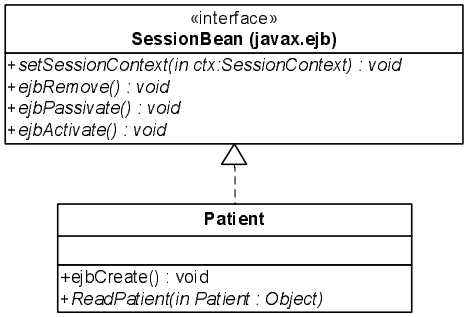


Figure 12 UML diagram of a basic session bean

The method `ejbCreate` is not defined within the `SessionBean` interface but is a key operation for custom session beans like the `PatienAgent` (Perrone & Chaganti, 2000). This is the method that is called by the container when it needs to create an instance of the bean, either when creating an initial pool of instance or to serve a client's request. The container calls the `ejbRemove` method when it needs to decommission the bean instance from handling more client requests (Perrone & Chaganti, 2000). Furthermore, for stateless session beans it is solely up to the container to determine when to call the remove method on a particular bean, in other words it is not bound in any way to the EJB client (Perrone & Chaganti, 2000).

According to Perrone and Chaganti (2000) the methods `ejbPassivate` and `ejbActivate` are used when a stateless bean is removed from active memory (passivated) and when it is brought back into active memory (activated).

The `ReadPatient` is an example of a method that might belong to the stateless session bean `Patient`. It is a method that is called from a client to load a patient from the database, an example of this is in Appendix B.

6.2 Dependability: Security

Cavanee and Keeton (2002) point out security goals in the EJB architecture: One is to lessen the burden for the bean provider when dealing with security issues. Another goal is to allow support for security policies that are set up by the developer rather then the bean provider. These goals are useful for Theriak, it is advantageous to have support from the technology to implement security, and it is advantageous to be able to adopt these security aspects to different healthcare organization environments.

Authentication in EJB is often accomplished by the use of JNDI API as described by Monson-Haefel (2001). He defines JNDI (Java Naming and Directory Interface) as something that supports various naming and directory services (a directory service is a more sophisticated naming service that organizes distributed objects into hierarchical structures and provides more sophisticated management features). This means that a client using JNDI can provide authentication information using the JNDI API to access a server.

A remote client is ready to use beans to accomplish some tasks as soon as it has been associated with a security identity. Monson-Haefel (2001) describes that the EJB server keeps track of each client and its identity, and when a client invokes a method on a component interface, the EJB server implicitly passes the client's identity with the method invocation. The EJB object or EJB home then receives the method invocation and checks the identity to ensure that the client is allowed to invoke that method.

According to the specification of EJB, handling security is divided into two methods called programmatic and declarative security (Cavanee & Keeton, 2002). With declarative security the security policy is completely external to the application code. Programmatic security is strongly coupled with the security environment which means that if a bean is deployed in another environment it is probably necessary to make changes in the source code.

6.2.1 Declarative security

To use declarative security, the system designer must define which security roles are allowed to access an enterprise bean. Deployment descriptors are used to define which logical roles are allowed to access which bean methods at runtime (Monson-Haefel, 2001). This means that the roles do not directly reflect the users or groups in a specific operational environment. Instead, when the bean is deployed the roles are mapped to real world users and groups. This allows portability, since the roles can be mapped to users and groups specific to the operational environment every time the bean is deployed in a new system. Security identity is represented by a `java.security.Principal` object. The `Principal` acts as a representative for

users, groups, organizations etc. to the EJB access-control architecture (Monson-Haefel, 2001).

Security roles are defined in the deployment descriptor as follows (Cavanee & Keeton, 2002):

```
...
    <ejb-name>PatientEJB</ejb-name>
        ...
        <security-role-ref>
            <description>Administrator</description>
            <role-name>Administrator</role-name>
        </security-role-ref>
        ...
...
```

Cavanee and Keeton (2002) point out that if no security roles are added to the deployment descriptor it is up to the deployer to understand the business methods in the enterprise beans and the operational environment to do the mapping.

Roles can be associated with methods with the `<method-permission>` tags. These tags contain one or more `<methods>` tags which specify what methods the role is associated with (Monson-Haefel, 2001). The following XML code shows where the "Administrator" is associated with all the methods in the Patient EJB. Then another logical role, "ReadOnly" specifies that the user with that role can use the methods "getName" and "getSSN" which return the patient's name and social security number, respectively.

```
<method-permission>
    <role-name>Administrator</role-name>
    <method>
        <ejb-name>PatientEJB</ejb-name>
        <method-name>*</method-name>
    </method>
</method-permission>
<method-permission>
    <role-name>ReadOnly</role-name>
    <method>
        <ejb-name>PatientEJB</ejb-name>
        <method-name>getName</method-name>
    </method>
    <method>
        <ejb-name>PatientEJB</ejb-name>
        <method-name>getSSN</method-name>
    </method>
</method-permission>
```

A XML deployment descriptor can describe more than one EJB and the tags used to specify the method permissions and logical security roles are specified in a special section of the deployment descriptor (Monson-Haefel, 2001). This allows for several beans to use the same security roles.

The security method described here is implicit. The container takes care of checking that a client can access only those methods for which it has permission (Monson-Haefel, 2001). The security identity, the `Principal`, is propagated with each method invocation from the client to the bean. If the client is a member of a role mapped to that method the method is invoked, otherwise an exception is raised. If a bean accesses any other enterprise bean while servicing a client, it will forward the security identity to that bean. This means that the security identity is propagated from one bean invocation to the next, ensuring that the access is controlled whether or not the method is invoked directly or not (Monson-Haefel, 2001).

A set of methods can be declared as unchecked, which means that they are not checked for permission before they are invoked and can thereby be accessed by any client, no matter what role it has (Monson-Haefel, 2001).

Depending on the tool used to develop the enterprise beans, security roles do not necessarily need to be implemented manually in the deployment descriptor. Tools like Borland JBuilder provide an interface to declare roles for enterprise beans.

6.2.2 Programmatic security

Programmatic security is not recommended by Cavanee and Keeton but situations might occur where this is necessary, like if the declarative method does not provide enough flexibility to fulfill a business need. It is not recommended because security roles would be defined in the source code which therefore would need to be changed in order to manage security roles.

6.3 Dependability: Availability and reliability

Reliability and availability were described in section 2.3 as the continuity of service and the readiness for usage. Although these are important concepts, it is hard to prove or measure those (Perrone & Chaganti, 2000). It is mostly up to the application server vendors to handle this (Jewell, 2000). Perrone and Chaganti (2000) describe that availability of service in application servers is primarily provided as a function of proper thread management, resource management, transaction management and state management. This chapter describes how clustering and exceptions are used to increase availability and reliability in EJB.

6.3.1 Clustering

Jewell (2000) points out different possibilities for application server vendors to implement clustered EJBs. He defines a cluster as a loosely coupled group of application servers that provide shared access to the services that each server hosts. The aim of the cluster is to balance resource requests, provide failover, and ensure high availability. This also results in a greater scalability. Failover is the ability for a request to have a high availability switchover to another server without the disruption of the service (Jewell, 2000). The goal of failover is to accomplish this without the client ever noticing. Failover is a difficult feature to implement and the most desirable in systems such as Theriak's.

Vendors have different locations for clustering logic for an EJB; the JNDI Naming Server, the container and the stub (Jewell, 2000). The JNDI naming server is the initial access point for clients to retrieve a stub of a session and vendors can replicate this naming service across nodes in the cluster in order to make EJBs highly available. Application servers can have their EJB container located on all the nodes in the cluster so that the replicated stub can reference the skeleton on a different node. The entire cluster is then represented through a single IP address or a DNS name, which means that clients all connects to the same IP address and are then re-routed as needed according to a load balance algorithm.

Containers can, according to Jewell (2000), provide some load balancing and failover logic. This applies to stateful session beans; if a container on a server is burdened with many requests for a particular stateful session bean it can forward the request to a counterpart on another server. Then after the load on the container has been reduced it can start servicing requests again. In this way, containers can provide minimal failover capability (Jewell, 2000). When a stateful session bean is being created a backup copy, a replica, is placed on another server in the cluster that is only used if the primary fails. If the primary fails the backup becomes the primary and creates another backup copy. This is an example of passive replication since the backup replica is only used if the primary fails (see section 2.1).

The stub is the first object created and accessed by a remote client and since stubs and skeletons are typically generated by EJB compilers at deployment time, this can be vendor specific (Jewell, 2000). This means that vendors can implement some load balancing and failover schemes directly in the stub. The primary service of the stub is to create or load beans on a remote server and on which server the bean is ultimately created does not matter. The create and find enterprise bean methods could load balance requests on different skeletons in the cluster.

The stub is the object that is instantiated by the skeleton and returned back to the client (Jewell, 2000). It can perform load balancing and failover but there are problems with this approach. If a client has created an entity bean it will only exist on the application server where it was created, and a stub that accesses an entity bean can not freely load balance its request to other servers in the cluster since the entity bean will only be active on one server. In other words, the stub is dependent on the server that it came from and is not free to load balance at will.

Since the application server in the architecture described in section 2.6 is stateless it will consist only of stateless session beans. This means that is does not matter to the client which application server in the cluster it connects to because they all have the same set of session beans. The client does not care which application server fulfils it request as long as it is processed. An example scenario is that the business layer sends a patient object that it wants to be stored in the database. The JNDI locates an application server to process this request; this involves finding the appropriate application server, that is, an application server that is not loaded with requests already. When an application server is found the request is forwarded to it and received by the persistence layer. This means that a session bean, the `Patient` bean, has been created or picked out from a pool, and that bean is now processing the request. If the patient object is successfully stored in the database it will return a confirmation that is sent to the client, otherwise an exception is raised. This exception

needs to be handled, either by sending an error message to the client or by starting a failover maneuver which means that another application server receives the original request from the JNDI and tries to process it.

6.3.2 Exceptions

As already discussed in section 5.3.2, exceptions can be used to handle faults and errors. If a method throws an error it is catched and dealt with, hopefully before it becomes a failure, either by rolling back a transaction or by displaying an error message to the user. Either way, by using defensive programming the system becomes more reliable. Defensive programming is important to systems such as Theriak's since users handling sensitive data must be able to know whether their method request was performed successfully or not.

Monson-Haefel (2000) describes three kinds of exception handling when dealing with persistence. These are application exception, runtime exceptions and subsystem exceptions.

Application exceptions include the standard EJB application exceptions which are `CreateException`, `FinderException`, `ObjectNotFoundException`, `DublicateKeyException`, and `RemoveException`. The names of these exceptions are relatively descriptive; `CreateException` is thrown when an entity EJB object cannot be created, `FinderException` reports a failure of finding an EJB object, `ObjectNotFoundException` is thrown by a finder method to indicate that an EJB object does not exist, `DuplicateKeyException` is thrown when an entity EJB object can not be created because another object exists with the same key, and `RemoveException` is thrown at an attempt to remove an EJB object when the container does not allow it to be removed (Sun Microsystems, 2002). All these exception are thrown from the appropriate methods to indicate that a business logic error has occurred (Monson-Haefel, 2001). Besides these standard exceptions it is possible to define custom exceptions that are specific for some business problems.

Runtime exceptions are exceptions that are thrown from the Java virtual machine and indicate that a fairly serious error has occurred (Monson-Haefel, 2001). Examples of these kinds of exceptions are the `NullPointerException` and `IndexOutOfBoundException`, which are handled by the container automatically and should therefore, according to Monson-Haefel (2001), not be handled inside a bean method. The callback methods (like `ejbCreate()` and `ejbActivate()`) throw the `EJBException` when a serious error occurs. This exception, according to Sun Microsystems (2002), is thrown by a bean instance to its container to indicate that an unexpected error occurred in a business method.

Subsystem exceptions are exceptions thrown by other subsystems (such as JNDI) and should be wrapped in the `EJBException` or an application exception and rethrown from that exception (Monson-Haefel, 2001).

7 COM+ and EJB side by side

This chapter compares COM+, discussed in chapter 5, and EJB, which is discussed in chapter 6. These are both component based technologies that can be used to build enterprise systems. They both provide a variety of services for the application programmer, allowing her to focus on the business logic and let the technology handle the low-level "rocket science".

However, these technologies take slightly different approaches; developers must consider which technology suits them best. This is not always an easy task. COM+ ships with Windows 2000 and Windows XP, it is a part of the operating system and therefore "free", meaning that if you buy Windows, you have COM+. EJB on the other hand needs to be purchased as an add-on to Windows. But this is also one of the great advantages of EJB; it is not operating system specific. The point in mentioning this is that when comparing these technologies we have access to the COM+ runtime system through the Component Services MMC, the one that is integrated into Windows. With EJB there are different vendors that provide different interfaces for administering beans, and as a tool in this report Borland JBuilder was used. Because of this, some of the discussion might be specific to JBuilder.

7.1 The architecture revisited

The 4-tier architecture is discussed in chapter 2.6. It is an architecture suggested by Theriak that the company intends to use in their system for healthcare organizations. Even though this rapport discusses this architecture as a case-study, it can be generalized and used in other multi-tier systems with similar requirements. This chapter provides a short summary of the architecture with the previous two chapters in mind. The different layers in the architecture are showed in the following figure:

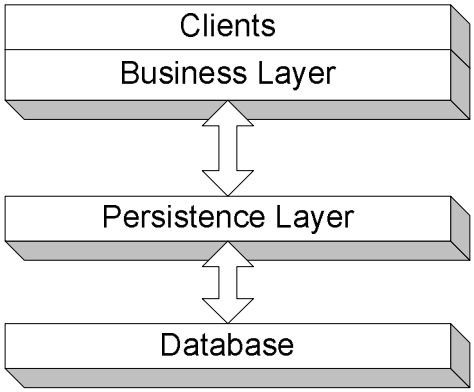


Figure 13 Different layers in the 4-tier architecture

The components reside in the persistence layer, the middleware. Their purpose is to provide services for the business layer. These services are in the form of storing and loading objects to and from the database. The components are stateless, which means that the component instances do not store any information. This improves scalability since the business layer can make use of any component, as long as it provides the interface that the business layer needs.

The reason for a multi-tier architecture like the one in Figure 13 is to provide a cohesive set of services by making each layer have its specific role. With this architecture a higher level of abstraction is reached by encapsulation. This means that each layer hides its implementation details from other layers.

Security is moved from the database to the application server, where the persistence layer resides. Each component is bound with some security constraints that define who are allowed to run which methods. This means that different clients can use the same connection to the database, since they do not explicitly have to authenticate themselves to the database. The persistence layer makes sure that clients are not allowed to run methods that they do not have permission for, and therefore a client can not manipulate data in the database that it is not allowed to, i.e. it can not access methods that manipulate this data.

This architecture provides many opportunities to increase scalability. By making the layers cohesive it is easier to spread them among several computers, in a distributed system. It should not be hard to add computers to the network of other computers that make up the application.

Scalability is very much up to the servers that are to be used. When using COM+ there are Microsoft servers that can be used, which are more specific to COM+ and the Microsoft platform, and when using EJB there are more choices, different server vendors providing different functionality, and application servers that are not necessarily bound to a specific operating system.

7.2 Enterprise Beans versus COM+ components

This chapter compares the COM+ components with the various types of Enterprise Beans.

7.2.1 Various Enterprise Bean types

As described in chapter 2.9 there are different kinds of enterprise beans; entity beans, session beans and message-driven beans. By definition they differ in functionality, but it is not always clear what kind of bean should be used. Working with a patient data, for example, clients would like to store information about the patient (like name and address), but they also might like to update the address. Now the question is whether to make the patient bean an entity bean or a session bean. According to Sessions (2000b), both are possible.

Monson-Haefel (2001) answers this question by saying that entity beans are developed to provide an interface to a set of data that defines a concept, whereas session beans access data and is usually read-only. With an entity bean the database key can be handled automatically in the container by the getPrimaryKey method. This, however, may cause undesirable performance overhead of the reference creation (Sessions, 2000b). This is because when creating a reference, the entity bean first looks up the primary key in the database, to make sure that it exists, then it must look in the database again for the data, for example the name of a patient.

7.2.2 Stateless session beans only

Furthermore, a stateless session bean could be used to list all patients that are taking a specific medicine. This requires that the bean joins the patient table and the medicine table in the database and sends its results to the business layer, the client. When this is done the bean is not necessarily destroyed; it is ready to serve the next client. It is possible to implement session beans to do the same job as entity beans, but by sending the database key with each method. This is the way it is done in COM+. The point of all this is that although EJB provides a variety of bean types, only one type would be preferable to use in the persistence layer, the stateless session bean.

7.2.3 Differences and similarities

COM+ does not provide such variety of component types and the COM+ stateless component model and the EJB stateless session beans are very similar. The session bean has a home interface which typically has a create method, and a remote interface which defines the methods that are then implemented in the bean class. The home interface is first called by the client and it returns a proxy to the remote interface which the client then uses. In COM+ there is only the interface and the class that implements that interface.

The implementation model for EJB is more attractive than for COM+. Making the bean creation code separate from the methods code is a more structural way than the approach in COM+. It could be possible to place the database connection code in the create method in the home interface so that when the client receives the proxy to the home interface the bean has established connection to the database (if the bean was instantiated and not taken from a pool of beans) and the client can start using the methods directly. In COM+, the instantiation code and the method code are in the same class so that the database connection is established when the client first calls that class (again, this is only when a component is instantiated and not received from a pool of components).

7.3 The persistence layer

The components that reside in the persistence layer service requests from a number of clients. Thus they have to be available and reliable, and provide security. This chapter discusses how the persistence layer can be realized using either COM+ or EJB.

In both COM+ and EJB, scalability is increased using stateless components and clustering technology. The clients connect to the cluster and are re-routed to the most available application server. In COM+ this is provided by the Clustering Service available in Microsoft Advanced Server and Datacenter Server. In EJB there are more choices. However, these choices require that Theriak goes into depth about what different application server vendors have to offer. Besides load balancing, the cluster needs to provide support for failover. Failover support is provided in Microsoft's Clustering Service and EJB vendors also have this support.

In chapter 5.1.2 the object-per-client model is described for COM+ components, where each client has its own component. This model can also be implemented in EJB, making each client have its own stateless session bean. Both COM+ and EJB provide support for pooling components so when clients are finished using a

component (or a bean) the component does not necessarily have to be destroyed, it can be placed in a pool so that the next client does not need to instantiate the component. This means, for example, that the next client does not have to instantiate a database connection because the component has already set up the connection when used by a previous client. Database connections are expensive so this increases the throughput in the system.

Security is important in the persistence layer, as has been discussed. COM+ and EJB both provide role-based security where it is possible to specify which roles are allowed to run which methods. This can be done both declaratively and programmatically, although the former is recommended since the latter requires that the roles are defined in the source code.
When specifying who are allowed to do what, it is done with roles, not users. It is possible to define that roles can do anything, or nothing, in the entire application, or it is possible to specify which roles can access with methods. In COM+ these roles are defined in the Component Services MMC, and then users are added to these roles. In EJB this is specified in the deployment descriptor, and JBuilder provides an interface to specify roles, but which roles are allowed to access which methods needs to be edited in the deployment descriptor. One great advantage that COM+ has is the authentication level (see section 5.2.1), where it is possible to define how secure the application needs to be.

In COM+ it is more comprehensible how the security is handled and using Component Services MMC makes it relatively easy to configure the security aspects (see figure in Appendix A). There it is possible to expand the methods available in an application and set each method's security properties. In EJB this is done in the deployment descriptor, as described in section 6.2.1, and is less understandable and maintainable.

Letting the persistence layer handle security also affects the scalability of the system. This is because clients can share connections to the database since they do not have to authenticate themselves explicitly to the database server. The persistence layer does not allow clients to access methods that manipulate data that the client is not supposed to be able to manipulate.

In this report reliability has been discussed from the point of view of defensive programming. This means handling errors and preventing failures. In COM+ this is dependent on the programming language that is used to implement the components. Enterprise beans can only be implemented with Java, and Java provides good exception handling. Visual Basic also provides exception handling although most of the real work is hidden from the programmer. To get the most out of error handling some other language is more suitable, like Visual C++.

However, comparing sections 5.3.2 Error handling and 6.3.2 Exceptions, it is clearer how exceptions are handled in Java. Visual C++ is a bit more difficult and requires more effort from the component developer.

Since a system such as Theriak's needs to be available when needed, clients should not have to wait for other clients to finish their work. Availability can be increased by using stateless components in the persistence layer. A client that needs a service

connects to the cluster and is routed to an available application server. A component or a bean is instantiated on the application server which is now ready to serve the client's request. Other clients connected to the application server have another instance of a component or a bean. This is possible since the application server is stateless; no state is shared between clients.

In Microsoft Corporation (2002b) some new features in COM+ are identified. These features are meant to improve scalability, availability and manageability (discussed in section 2.3) of COM+ applications. COM+ partitions are a feature that allows multiple versions of an application to be installed on the same machine. In other words, each partitions acts as a virtual server, containing some version of the application. This reduces the cost- and time-consuming efforts of managing multiple servers for different versions of an application.

Application recycling increases the overall stability of an application. This is a solution to allow COM+ applications to gracefully shut down a process and restart it. This is done because application performance can degrade over time because of memory leaks, reliance on third party code, and nonscalable resource usage.

Both technologies do very well in supporting scalability and dependability, and there is in fact little difference between them. The basic ideas for increasing scalability and dependability are similar in both COM+ and EJB.

8 Conclusions

This chapter concludes the work done in previous chapters and gives a recommendation for Theriak about whether to use COM+ or EJB to build their 4-tier distributed system. It also points out the project contributions, related work and ideas for future work.

8.1 Results and discussion

As can be seen from chapter 7 which discusses how COM+ and EJB can be used to realize the persistence layer, considering dependability and scalability aspects, the two technologies are very similar. This makes it difficult to base the recommendation entirely on how the technologies handle the criteria. There are other aspects, which are more common, that also influence the recommendation. Mostly, these aspects are related to language- and operating system neutrality.

The only programming language that can be used to implement EJB is Java. If a company like Theriak has been developing software in Java, and it knows and likes the language, EJB can be considered the way to go. However, if the company has not been using Java, COM+ provides a large variety of other possible programming languages to use. These are languages like Visual Basic, Delphi, Visual C++ and the latest language, C# which recently shipped with the .NET framework. Theriak has been developing their product in Delphi and by using COM+ they can continue to use Delphi, making it unnecessary for the employees to study a new programming language.

Portability and performance are two aspects that differ in EJB and COM+ (Sessions, 2000a). EJB aims for high portability, and to achieve portability the code is divided into two layers; the portability layer (which should be as thin as possible) and the operating system independent (OSI) layer (which should be as thick as possible). This means that programmers never use the operating system directly; they always request operating system services indirectly through the OSI, which translates the request into something that the operating system understands. Portability is always a great advantage, giving the possibility to leverage the system on a wide variety of operating systems.

Portability introduces more choices of application servers and it is even possible to use servers from different vendors simultaneously, although that would make the system as a whole more complicated. Portability also makes the system more flexible and provides the opportunity to change vendors later on, depending on the development in the hardware and software industry. One thing to consider is that there is actually no guarantee that EJB will work identically on different servers (from different vendors), there is always a chance that some minor changes need to be made if using different vendors.

Portability can also affect the cost of the system. Different vendors provide different application servers and these servers can vary in cost. This is a market where various vendors compete, which differs from COM+ where Microsoft is dominant.

In COM+ the approach is the opposite, there is no portability layer and programmers directly leverage the underlying operating system services (Sessions, 2000a). This reduces the overall system cost.

COM+ provides better performance under Windows than EJB because there is no portability layer that needs to translate the requests for the operating system. The operating system services are used directly. In the case of Theriak, they implement their product in a Windows environment and although portability is an advantage, performance is considered more important.

The recommendation for Theriak is to use COM+ to build their 4-tier distributed system. The main arguments, besides those in chapter 5 and 7, are the choice of programming language and better performance than EJB in a Windows environment. Besides, COM+ has a longer history and is a more established technology, a more mature technology that has proven itself in critical systems (Sessions, 2000b).

8.2 Contributions

The purpose of this chapter is to discuss the contributions of this project. The goal of the project is to provide a recommendation for a company like Theriak about whether to build a 4-tier distributed systems using COM+ or EJB. (The problem was defined in chapter 3, COM+ was discussed in chapter 5 and EJB in chapter 6. The two technologies were then discussed side by side in chapter 7, all this in the light of certain criteria; security, scalability, availability and reliability.)

What we have contributed with this project is a study in two component based technologies. This study is specific because it looks at a particular criteria and architecture. We have looked into how these technologies support these criteria independently and then compared them side by side.

There are two main contributions:

- A study of dependability support by two component based technologies.
    - This involves a study of security, availability and reliability aspects and how they can be increased.
    - A study of how scalability can be increased in both technologies.
- A suggestion for a 4-tier component-based architecture to be used in healthcare organizations with high dependability demands.
    - This involves a discussion about the technologies and how the persistence layer can be realized.

8.3 Future work

This chapter covers some ideas of future work, what is possible to study from the work done in this project.

This project focuses on the persistence layer in the application server. Next step could be to move the focus up to the business layer in the architecture. There it is possible to look at how the business layer handles the response from the application server and displays it to the user. This involves how the business layer decodes the XML data that it receives from the application server.

Another possibility is to thoroughly study how the communication between the business layer and the application layer, which is encoded in XML, can be accomplished. This could be done from a COM+ friendly programming language and security issues in local area networks or the Internet could be further studied.

A more practical approach can be taken, meaning designing and implementing a prototype in either COM+ or EJB (even both). After the implementation of the prototype it could be possible to perform various tests in order to strengthen arguments about which technique is more appropriate in a 4-tier distributed system. These could be, for example, security tests or testing scalability.

There is another technology called CORBA which is a middleware design that irrespective of programming language, hardware, software and network allows application programs to communicate with each other (Coulouris et al., 2001). There is also a new specification called CORBA Component Model (CCM) which introduces an infrastructure for components much like COM+ and EJB (Sessions, 2000b). It is a possibility to study CCM just as COM+ and EJB have been studied in this project.

.NET is a new framework from Microsoft and provides some new services and components called .NET components or assemblies. It is possible to study the .NET framework and evaluate what it has to offer on top of COM+.

It is always possible to go deeper and deeper in researching the criteria used. A possibility is to extend the study here with other criteria. These could be criteria like maintainability and flexibility.

Acknowledgements

I would like to thank my supervisor, Sanny Gustavsson, for his guidance and valuable advices and opinions regarding the project. I would also like to thank my examiner, Mikael Berndtsson, for his reviews.

Ægir Leifsson, senior developer at Theriak who had the original idea for this project, thank you for allowing me to carry out this project and answer all my questions. It has been an adventure and a great challenge.

Many thanks to my family and friends in Iceland for their loving support and understanding, without you this project would not exist.

References

Ambler, S. W. (1999) *The design of a robust persistence layer for relational databases*. Ambysoft Inc. Available at Internet: http://www.ambysoft.com/persistenceLayer.pdf [Accessed 02.02.10].

Brown, R., Baron, W. & Chadwick III, W. D. (2001) *Designing solutions with COM+ technologies*. USA: Microsoft Press.

Coulouris, G., Dollimore, J. & Kindberg, T. (2001) *Distributed systems concepts and design* (Third edition). USA: Pearsson Education Limited.

Ewald, T. (2001) *Transactional COM+: Building scalable applications*. USA: Addison-Wesley.

Fowler, M. & Scott, K. (2000) *UML distilled: A brief guide to the standard object modeling language* (Second edition). USA: Addison-Wesley.

Freeze, W. S. (2000) *Visual basic developer's guide to COM and COM+*. USA: Sybex Inc.

Jewell, T. (2000) *EJB 2 clustering with application servers*. O'Reilly Network. Available at Internet: http://www.onjava.com/pub/a/onjava/2000/12/15/ejb_clustering.html [Accessed 02.04.12].

Laprie, J.C. (ed.) (1994) *Dependability: basic concepts and terminology*. Springer Verlag.

Löwy, J. (2001) *COM and .NET component services*. USA: O'Reilly & Associates, Inc.

Microsoft Corporation. (1995) *The COM specification.* Available at Internet: http://www.microsoft.com/com/resources/comdocs.asp [Accessed 02.02.14].

Microsoft Corporation. (2002a) *The component object model*. Available at Internet: http://msdn.microsoft.com/library/en-us/com/com_757w.asp [Accessed 02.02.14].

Microsoft Corporation. (2002b) *What's new in COM+*. Available at Internet: http://msdn.microsoft.com/library/en-us/cossdk/htm/whatsnewcomplus_350z.asp [Accessed 02.02.18].

Microsoft Corporation. (2002c) *Windows 2000 Clustering Technologies: Cluster Service Architecture*. Available at Internet: http://www.microsoft.com/windows2000/docs/ClusterArch.doc [Accessed 02.04.22].

Mojica, J. (2001) *COM+ programming with Visual Basic*. USA: O'Reilly & Assiciates, Inc.

Monson-Haefel, R. (2001) *Enterprise JavaBeans* (Third edition). USA: O'Reilly & Associates, Inc.

Mullender, S. (ed.) (1993) *Distributed systems* (Second edition). England: ACM Press.

Perrone, P. J. & Chaganti, V. S. R. K. R. (2000) *Building Java enterprise systems with J2EE*. USA: Sams Publising.

Pressman, R. S. (2000) *Software Engineering: a practitioner's approach* (Fifth edition). UK: McGraw-Hill International Limited.

Sessions, R. (2000a) EJB vs. COM+. *Software Magazine, 20(5)*, 18-19.

Sessions, R. (2000b) *COM+ and the battle for the middle tier*. USA: John Wiley & Sons, Inc.

Sun Microsystems. (2001) *Enterprise JavaBeans™ Specification, version 2.0*. Availible at Internet: ftp://ftp.java.sun.com/pub/ejb/947q9tbb/ejb-2_0-fr2-spec.pdf [Accessed 02.02.27].

Sun Microsystems. (2002) *Enterprise JavaBeans 2.0 Documentation*. Availible at Internet: http://java.sun.com/products/ejb/javadoc-2_0-fr/ [Accessed 02.04.11].

Appendix A: An example of a COM+ component

In this appendix there is an example of a small COM+ component. This is a component named Hospital, and demonstrates how to get a patient name from a database.

The class IHospital defines the interface of the `Hospital` class:

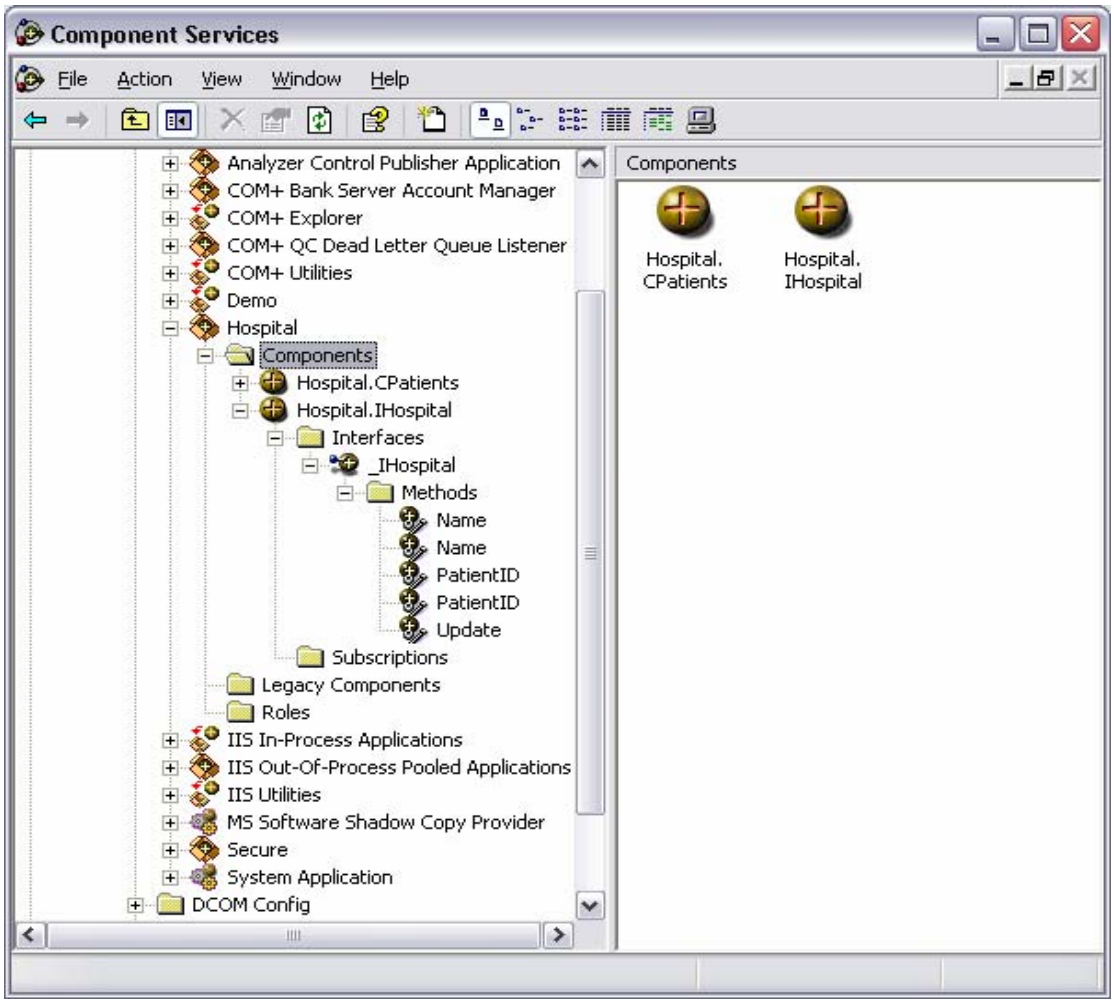
```
'Class IHospital

Public Property Get Name() As String
End Property
```

This class does not provide any implementation of the methods, it is up to the next class called `CPatients`.

The class `CPatients` implements these methods, in addition it has an initialize method that gets a connection to the database, and a finalize method that closes the connection:

```
‘Class CPatients, implements IHospital

Implements IHospital
Dim rs As Recordset

Private Sub Class_Initialize()
    Set rs = New Recordset

    rs.ActiveConnection = ‘Connection string to database

    ‘select the record from the patient table in the database
    rs.Open "select * from Patient"
End Sub

Private Sub Class_Terminate()
    rs.Close
    Set rs = Nothing
End Sub

Private Property Get IHospital_Name() As String
    IHospital_Name = rs("Name")
End Property
```

Now the interface has been defined and implemented, and it is possible to build a .DLL file from these classes. That file is inserted into the COM+ runtime environment through the Component Services MMC. The following figure shows the component.



From this window it is possible to manage the COM+ runtime environment services.

A simple example of a client that uses this component is illustrated in the following code:

```
Private Sub btnGetPatient_Click()
    Dim pat As IHospital
    Set pat = New CPatients

    MsgBox ("Patient name: " & pat.Name)

    Set pat = Nothing
End Sub
```

This client displays a message box containing the name of the patient when a button is pressed.

Appendix B: An example of a session bean

The following is an example of an implementation of a session bean according to the model in Figure 12.

The class `PatientHome` defines the create method of the `Patient`:

```
public interface PatientHome extends javax.ejb.EJBHome {
  public    Patient    create()    throws    CreateException,
RemoteException;
}
```

The class `Patient` defines the methods that are available, in this case there is only one:

```
public interface Patient extends javax.ejb.EJBObject {
  public void ReadPatient() throws RemoteException;
}
```

The class `PatientBean` is the class that implements these interfaces:

```
public class PatientBean implements SessionBean {

  SessionContext sessionContext;
  Database database = new Database();

  public void ejbCreate() throws CreateException {
  }

  public void ejbRemove() {
  }

  public void ejbActivate() {
  }

  public void ejbPassivate() {
  }

  public void setSessionContext(SessionContext sessionContext)
  {
    this.sessionContext = sessionContext;
  }

  public PatientBean() {
    try {
      //initialize database connection;
    }
    catch Exception
  }
```

```
  public void ReadPatient() {
    try {
      Statement stat = database.createStatement();
      ResultSet   rs   =   stat.executeQuery("SELECT   *   FROM
PATIENT");
      System.out.println(rs.getString(1)+" "+rs.getString(2));
    }
    catch (SQLException e) {
      System.out.println(e.toString());
    }
  }
}
```

As can be seen there is not much implementation in this example, but it gives a general idea of how stateless session beans look like.